conference

proceedings

Proceedings of LISA '11: 25th Large Installation System Administration Conference

Boston, Massachusetts, December 4–9, 2011

SPONSORED BY

in cooperation with

LOPSA

LISA '11: 25th Large Installation System Administration Conference

Boston, Massachusetts
December 4–9, 2011

© 2011 by The USENIX Association
All Rights Reserved

ISBN 978-931971-88-1

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings.

USENIX acknowledges all trademarks herein.

USENIX Association

Proceedings of LISA ’11: 25th Large Installation System Administration Conference

December 4–9, 2011

Boston, Massachusetts

Conference Organizers

Program Co-Chairs
Thomas A. Limoncelli, *Google, Inc.*
Doug Hughes, *D. E. Shaw Research, LLC*

Program Committee
Narayan Desai, *Argonne National Lab*
Andrew Hume, *AT&T Labs—Research*
Duncan Hutty, *ZOLL Medical Corporation*
Dinah McNutt, *Google, Inc.*
Tim Nelson, *Worcester Polytechnic Institute*
Mario Obejas, *Raytheon*
Mark Roth, *Google, Inc.*
Carolyn Rowland, *National Institute of Standards and Technology (NIST)*
Federico D. Sacerdoti, *Aien Capital & Aien Technology*
Marc Stavely, *Consultant*
Nicole Forsgren Velasquez, *Pepperdine University*
Avleen Vig, *Etsy, Inc.*
David Williamson, *Microsoft Tellme*

Invited Talks Coordinators
Æleen Frisch, *Exponential Consulting*
Kent Skaar, *VMware, Inc.*

Workshops Coordinator
Cory Lueninghoener, *Los Alamos National Laboratory*

Guru Is In Coordinator
Chris St. Pierre, *Oak Ridge National Laboratory*

Poster Session Coordinator
Matt Disney, *Oak Ridge National Laboratory*

Work-in-Progress Reports (WiPs) Coordinator
William Bilancio, *Arora and Associates, P.C.*

Training Program
Daniel V. Klein, *USENIX Association*

USENIX Board Liaison
David N. Blank-Edelman, *Northeastern University*

Steering Committee
Paul Anderson, *University of Edinburgh*
David N. Blank-Edelman, *Northeastern University*
Mark Burgess, *CFEngine*
Alva L. Couch, *Tufts University*
Rudi van Drunen, *Competa IT*
Æleen Frisch, *Exponential Consulting*
Xev Gittler, *Morgan Stanley*
William LeFebvre, *Digital Valence, LLC*
Mario Obejas, *Raytheon*
Ellie Young, *USENIX Association*
Elizabeth Zwicky, *Consultant*

The USENIX Association Staff

External Reviewers

Paul Armstrong
Derek J. Balling
Steve Barber
Matthew Barr
Lois Bennett
Ken Breeman
Travis Campbell
Brent Chapman
Marc Chiarini
Alva L. Couch
Matt Disney
Rudi van Drunen
Bill Lefebvre
Cory Lueninghoener
Chris McEniry
Adam Moskowitz
Mario Obejas
Tobias Oetiker
Cat Okita
Eric Radman
Benoit Sigoure
Josh Simon
Kent Skaar
Ozan Yigit

LISA ’11:
25th Large Installation System Administration Conference
December 4–9, 2011
Boston, Massachusetts

Message from the Program Co-Chairs. vii

Wednesday, December 7

Perspicacious Packaging

Staging Package Deployment via Repository Management. 1
Chris St. Pierre and Matt Hermanson, Oak Ridge National Laboratory

CDE: Run Any Linux Application On-Demand Without Installation 9
Philip J. Guo, Stanford University

Improving Virtual Appliance Management through Virtual Layered File Systems 25
Shaya Potter and Jason Nieh, Columbia University

Clusters and Configuration Control

Sequencer: Smart Control of Hardware and Software Components in Clusters (and Beyond) 39
Pierre Vignéras, Bull, Architect of an Open World

Automated Planning for Configuration Changes 57
Herry Herry, Paul Anderson, and Gerhard Wickler, University of Edinburgh

Fine-grained Access-control for the Puppet Configuration Language 69
Bart Vanbrabant, Joris Peeraer, and Wouter Joosen, DistriNet, K.U. Leuven

Security 1

Tiqr: A Novel Take on Two-Factor Authentication. 81
Roland M. van Rijswijk and Joost van Dijk, SURFnet BV

Building Useful Security Infrastructure for Free (Practice & Experience Report) 99
Brad Lhotsky, National Institutes on Health, National Institute on Aging, Intramural Research Program

Local System Security via SSHD Instrumentation. 109
Scott Campbell, National Energy Research Scientific Computing Center, Lawrence Berkeley National Lab

Thursday, December 8

From Small Migration to Big Iron

Adventures in (Small) Datacenter Migration (Practice & Experience Report). 121
Jon Kuroda, Jeff Anderson-Lee, Albert Goto, and Scott McNally, University of California, Berkeley

Bringing Up Cielo: Experiences with a Cray XE6 System, or, Getting Started with Your New 140k Processor System (Practice & Experience Report) 131
Cory Lueninghoener, Daryl Grunau, Timothy Harrington, Kathleen Kelly, and Quellyn Snead, Los Alamos National Laboratory

Backup Bonanza

Capacity Forecasting in a Backup Storage Environment (Practice & Experience Report). 141
Mark Chamness, EMC

Content-aware Load Balancing for Distributed Backup 151
Fred Douglis and Deepti Bhardwaj, EMC; Hangwei Qian, Case Western Reserve University; Philip Shilane, EMC

To the Cloud!

Getting to Elastic: Adapting a Legacy Vertical Application Environment for Scalability 169
Eric Shamow, Puppet Labs

Scaling on EC2 in a Fast-Paced Environment (Practice & Experience Report) 179
Nicolas Brousse, TubeMogul, Inc.

Honey and Eggs: Keeping Out the Bad Guys with Food

DarkNOC: Dashboard for Honeypot Management 189
Bertrand Sobesto and Michel Cukier, University of Maryland; Matti Hiltunen, Dave Kormann, and Gregg Vesonder, AT&T Labs Research; Robin Berthier, University of Illinois

A Cuckoo's Egg in the Malware Nest: On-the-fly Signature-less Malware Analysis, Detection, and Containment for Large Networks 201
Damiano Bolzoni and Christiaan Schade, University of Twente; Sandro Etalle, University of Twente and Eindhoven Technical University

Seriously Snooping Packets

Auto-learning of SMTP TCP Transport-Layer Features for Spam and Abusive Message Detection. 217
Georgios Kakavelakis, Robert Beverly, and Joel Young, Naval Postgraduate School

Using Active Intrusion Detection to Recover Network Trust 227
John F. Williamson and Sergey Bratus, Dartmouth College; Michael E. Locasto, University of Calgary; Sean W. Smith, Dartmouth College

Friday, December 9

Network Security

Community-based Analysis of Netflow for Early Detection of Security Incidents 241
Stefan Weigert, TU Dresden; Matti A. Hiltunen, AT&T Labs Research; Christof Fetzer, TU Dresden

WCIS: A Prototype for Detecting Zero-Day Attacks in Web Server Requests 253
Melissa Danforth, California State University, Bakersfield

Networking 1

Automating Network and Service Configuration Using NETCONF and YANG 267
Stefan Wallin, Luleå University of Technology; Claes Wikström, Tail-f Systems AB

Deploying IPv6 in the Google Enterprise Network: Lessons Learned 281
Haythum Babiker, Irena Nikolova, and Kiran Kumar Chittimaneni, Google

Experiences with BOWL: Managing an Outdoor WiFi Network (or How to Keep Both Internet Users and Researchers Happy?) (Practice & Experience Report) 287
T. Fischer, T. Hühn, R. Kuck, R. Merz, J. Schulz-Zander, and C. Sengul, TU Berlin/Deutsche Telekom Laboratories

Migrations, Mental Maps, and Make Modernization

Why Do Migrations Fail and What Can We Do about It? 293
Gong Zhang and Ling Liu, Georgia Institute of Technology

Provenance for System Troubleshooting 311
Marc Chiarini, Harvard SEAS

Debugging Makefiles with remake 323
Rocky Bernstein

Message from the Program Co-Chairs

Dear LISA ’11 Attendee,

There are two kinds of LISA attendees: those who read this letter at the conference and those who read it after they’ve returned home. To the first group, get ready for six days of brain-filling, technology-packed, geek-centric tutorials, speakers, papers, and more! To those that are reading this after the conference, we ask, “What’s it like living in the future? How was the conference? What cool tips and tools did you take home with you to make your job easier?”

Being a sysadmin is kind of like living in the future. You work with technology every day that would make Buck Rogers jealous. Most of our friends are jealous, too. When LISA started 25 years ago, a “large site” had 10 computers, each the size of a dishwasher, with a few gigabytes of combined storage. Today our cell phones have 32GB of “compact flash,” which is often more than the NFS quota we give our users.

Attending LISA is kind of like spending a week living in the future. We learn technologies that are cutting-edge—little known now, but next year everyone will be talking about them. When we return from LISA we sound like time travelers visiting from the future talking about new and futuristic stuff. LISA makes us look good.

LISA rarely has a cohesive conference theme, but this year we thought it was important to highlight DevOps, as it is a significant cultural change. Although DevOps is often thought of as “something big Web sites do,” the lessons learned transfer well to enterprise computing.

LISA has always been assembled using the sweat of many dedicated volunteers. It takes a lot of effort to put a conference like this together, and this year is no different. Most prominent are the Invited Talks committee (Æleen Frisch and Kent Skaar) and the Program Committee (Narayan Desai, Andrew Hume, Duncan Hutty, Dinah McNutt, Tim Nelson, Mario Obejas, Mark Roth, Carolyn Rowland, Federico D. Sacerdoti, Marc Stavely, Nicole Forsgren Velasquez, Avleen Vig, and David Williamson), but also important are the Workshops Coordinator (Cory Lueninghoener), the Guru Is In Coordinator (Chris St. Pierre), the Poster Session Coordinator (Matt Disney), and the Work-in-Progress Reports Coordinator (William Bilancio). We couldn’t have done it without every one of them. Of course, nothing would happen without the leadership of the USENIX staff. We are indebted to you all!

Of the 63 papers submitted, we accepted 28. These papers represent the best “deep thought” research, as well as Practice and Experience Reports that tell the stories from people “in the trenches.” We encourage you to read them all. However, the power of LISA is the personal interaction: introduce yourself to the attendees standing in line near you, strike up a conversation with the person sitting next to you. And remember to have fun!

Sincerely,
Thomas A. Limoncelli, *Google, Inc.*
Doug Hughes, *D. E. Shaw Research, LLC*
Program Co-Chairs

Staging Package Deployment via Repository Management

Chris St. Pierre - stpierreca@ornl.gov
Matt Hermanson - mjhermanson@ornl.gov
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN, USA*

Abstract

This paper describes an approach for managing package versions and updates in a homogenous manner across a heterogenous environment by intensively managing a set of software repositories rather than by managing the clients. This entails maintaining multiple local mirrors, each of which is aimed at a different class of client: One is directly synchronized from the upstream repositories, while others are maintained from that repository according to various policies that specify which packages are to be automatically pulled from upstream (and therefore automatically installed without any local vetting) and which are to be considered more carefully – likely installed in a testing environment, for instance – before they are deployed widely.

Background

It is important to understand some points about our environment, as they provide important constraints to our solution.

We are lucky enough to run a fairly homogenous set of operating systems consisting primarily of Red Hat Enterprise Linux and CentOS servers, with fair numbers of Fedora and SuSE outliers. In short, we are dealing entirely with RPM-based packaging, and with operating systems that are capable of using yum [12]. As yum is the default package management utility for the majority of our servers, we opted to use yum rather than try to switch to another package management utility.

For configuration management, we chose to use Bcfg2 [3] for reasons wholly unrelated to package and software management. Bcfg2 is a Python and XML-based configuration management engine that "helps system administrators produce a consistent, reproducible, and verifiable description of their environment" [3]. It is in particular the focus on reproducibility and verification that forced us to consider updating and patching anew.

In order to guarantee that a given configuration – where a "configuration" is defined as the set of paths, files, packages, and so forth, that describes a single system – is fully replicable, Bcfg2 ensures that every package specified for a system is the latest available from that system's software repositories [8]. (As will be noted, this can be overridden by specifying an explicit package version.) This grants the system administrator two important abilities: to provision identical machines that will remain identical; and to reprovision machines to the exact same state they were previously in. But it also makes it unreasonable to simply use the vendor's software repositories (or other upstream repositories), since all updates will be installed immediately without any vetting. The same problem presents itself even with a local mirror.

Bcfg2 can also use "the client's response to the specification ... to assess the completeness of the specification" [3]. For this to happen, the Bcfg2 server must be able to understand what a "complete" specification entails, and so the server does not entirely delegate package installation to the Bcfg2 client. Instead, it performs package dependency resolution on the server rather than allowing the client to set its own configuration. This necessitates ensuring that the Bcfg2 Packages plugin uses the same

*This paper has been authored by contractors of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

yum configuration as the clients; Bcfg2 has support for making this rather simple [8], but the Packages plugin does not support the full range of yum functionality, so certain functions like the "versionlock" plugin and even package excludes, are not available. Due to the architecture of Bcfg2 – architecture designed to guarantee replicability and verification of server configurations – it is not feasible or, in most cases, possible to do client-based package and repository management. This became critically important in selecting a solution.

Other Solutions

There are a vast number of potential solutions to this problem that would seem to be low-hanging fruit – far simpler to implement, at least initially, than our ultimate solution – but that would not work, for various reasons.

Yum Excludes

A core yum feature is the ability to exclude certain packages from updates or installation [13]. At first, this would seem to be a solution to the problem of package versioning: simply install the package version you want, and then exclude it from further updates. But this has several issues that made it unsuitable for our use (or, we believe, this use case in general):

- It does not (and cannot) guarantee a specific version. Using excludes to set a version depends on that version being installed (manually) prior to adding the package to the exclude list.

- There is no guarantee that the package is still in the repository. Many mainstream repositories[1] do not retain older versions in the same repository as current packages. Consequently, when reinstalling a machine where yum excludes have been used to set package versions (or when attempting to duplicate such a machine), there is no guarantee that the package version expected will even be available.

- In order to use yum excludes to control package versions, a very specific order of events must occur: first, the machine must be installed without the target package included (as Kickstart, the RHEL installation tool, does not support installing a specific version of a package [1]); next, the correct package version must be installed; and finally, the package must be added to the exclude list. If this happens out of order, then the wrong version of the package might be installed, or the package might not be installed at all.

- Supplying a permitted update to a package is even more difficult, as it involves removing the package exclusion, updating to the correct version, and then restoring the exclusion. A configuration management system would have to have tremendously granular control over the order in which actions are performed to accomplish this delicate goal.

- As discussed earlier, Bcfg2 performs dependency resolution on the server side in order to provide a guarantee that a client's configuration is fully specified. By using yum excludes – which cannot be configured in Bcfg2's internal dependency resolver – the relationship between the client and the server is broken, and Bcfg2 will in perpetuity claim that the client is out of sync with the server, thus reducing the usefulness of the Bcfg2 reporting tools.

While yum excludes appear at first to be a viable option, their use to set package versions is not replicable, consistent, and cannot be trivially automated.

Specifying Versions in Bcfg2

Bcfg2 is capable of specifying specific versions of packages in the specification, e.g.:

```
<BoundPackage name="glibc" type="yum">
  <Instance version="2.13" release="1"
            arch="i686"/>
  <Instance version="2.13" release="1"
            arch="x86_64"/>
</BoundPackage>
```

This is obviously quite verbose (more so because the example uses a multi-arch package), and as a result of its verbosity it is also error-prone. Having to recopy the version, release, and architecture of a package – separately – is not always a trivial process, and the relatively few constraints of version and release strings makes it less so. For instance, given the package:

```
iomemory-vsl-2.6.35.12-88.fc14.x86_64-
    2.3.0.281-1.0.fc14.x86_64.rpm
```

The package name is "iomemory-vsl-2.6.35.12-88.fc14.x86_64" (which refers to the specific kernel for which it was built), the version is "2.3.0.281" and the release is "1.0.fc14".[2] This can be clarified through use of the `--queryformat` option to `rpm`, but the fact that more advanced RPM commands are necessary makes it clear that this approach is untenable in general. Even more worrisome is the package epoch, a sort of "super-version," which RPM cleverly hides by default, but could cause a newer package to be installed if it was not specified properly.

Maintenance is also tedious, as it involves endlessly updating verbose version strings; recall that a given version is just shorthand for what we actually care about – that a package *works*.

This approach also does not abrogate the use of yum on a system to update it beyond the appropriate point. The only thing keeping a package at the chosen version is Bcfg2's own self-restraint; if an admin on a machine lacks that same self-restraint, then he or she could easily update a package that was not to be updated, whereupon Bcfg2 would try to downgrade it.

Finally, this approach presents specific difficulties for us, as our adoption of Bcfg2 is far from complete; large swaths of the center still use Cfengine 2, and some machines – particularly compute and storage platforms – operate in a diskless manner and do not use configuration management tools in a traditional manner. They depend entirely on their images for package versions, so specifying versions in Bcfg2 would not help.

To clarify, using Bcfg2 forced us to reconsider this problem, and any solution must be capable of working with Bcfg2, but it cannot be assumed that the solution may leverage Bcfg2.

Yum versionlock

Using yum's own version locking system would appear to improve upon pegging versions in Bcfg2: it works on all systems, regardless of whether or not they use Bcfg2; and a shortcut command, `yum versionlock <package-name>`, is provided to make the process of maintaining versions less error-prone.[3]

It also solves many of the problems of yum excludes, but suffers from a critical flaw in that approach: by setting package versions on the client, the relationship between the Bcfg2 client and server would be broken.

Combinations of these three approaches merely exhibit combinations of their flaws. For instance, the promising combination of yum's versionlock plugin and specifying the version in Bcfg2 would ensure that the Bcfg2 client and server were of a mind about package versions, and would work on non-Bcfg2 machines; however, it would forfeit versionlock's ease of use and require the administrator to once again manually copy package versions.

Spacewalk

Spacewalk was the first full-featured solution we looked at that aims to replace the mirroring portion of this relationship; all of the other potential solutions listed thus far have attempted to work with a "dumb" mirror and use yum features to work around the problem we have described. Spacewalk is a local mirror system that "manages software content updates for Red Hat derived [*sic*] distributions" [10]; it is a tremendously full-featured system, with support for custom "channels," collections of packages assembled in an ad-hoc basis.

Unfortunately, Spacewalk was a non-starter for us for the same reason that it has failed to gain much traction in the community at large: of the two versions of Spacewalk, only the Oracle version actually implements all of the features; the PostgreSQL version is deeply underfeatured, even after several years of work by the Spacewalk team to port all of the Oracle stored procedures.

As it turns out, Red Hat has a successor in mind for Spacewalk and Satellite: CloudForms [14]. The content management portion of CloudForms – roughly corresponding to the mirror and repository management functionality of Spacewalk – is Pulp.

A solution: Pulp

Pulp is a tool "for managing software repositories and their associated content, such as packages, errata, and distributions" [7]. It is, as noted, the spiritual successor to Spacewalk, and so implements the vast majority of Spacewalk's repository management features without the dependency on Oracle.

Pulp's usage model involves syncing multiple upstream repositories locally; these repositories can then be *cloned*, which uses hard links to sync them locally with almost no disk space used. This allows us to sync a repository once, then duplicate it as many times as necessary to support multiple teams and multiple stability levels. The sync process supports *filters*, which allow us to blacklist or whitelist

packages and thus exclude "impactful" packages from automatic updates.

Pulp also supports manually adding packages to and removing packages from repositories, so we can later update a given package across all machines that use a repository with a single command. Adding and removing also tracks dependencies, so it's not possible to add a package to a repository without adding the dependencies necessary to install it.[4]

Workflow

Pulp provides us with the framework to implement a solution to the problem outlined earlier, but even as featureful as it is it remains a fairly basic tool. Our workflow – enforced by the features Pulp provides, by segregating repositories, by policy, and by a nascent in-house web interface – provides the bulk of the solution. Briefly, we segregate repositories by tier to test packages before site-wide roll-outs, and by team to ensure operational separation. Packages are automatically synced between tiers based on package filters, which blacklist certain packages that must be promoted manually. This ensures that most packages benefit from up to two weeks of community testing before being deployed site-wide, and packages that we have judged to be more potentially "impactful" from more focused local testing as well.

Tiered Repositories

We maintain different repository sets for different "levels" of stability. We chose to maintain three tiers:

`live` Synced daily from upstream repositories; not used on any machines, but maintained due to operational requirements within Pulp[5] and for reference.

`unstable` Synced daily from `live`, with the exception of selected "impactful" packages (more about which shortly), which can be manually promoted from `live`.

`stable` Synced daily from `unstable`, with the exception of the same "impactful" packages, which can be manually promoted from `unstable`.

This three-tiered approach guarantees that packages in `stable` are at least two days old, and "impactful" packages have been in testing by machines using the `unstable` branch. When a package is released from upstream and sync to public mirrors, those packages are pulled down into local repositories. From then on the package in under the control of Pulp. Initially, a package is considered unstable and is only deployed to those systems that look at the repositories in the `unstable` tier. After a period of time, the package is then promoted into the `stable` repositories, and thus to production machines.

In order to ensure that packages in `unstable` receive ample testing before being promoted to `stable`, we divide machines amongst those two tiers thusly:

- All internal test machines – that is, all machines whose sole purpose is to provide test and development platforms to customers within the group – use the `unstable` branch. Many of these machines are similar, if not identical, to production or external test machines.
- Where multiple identical machines exist for a single purpose, whether in an active-active or active-passive configuration, exactly one machine will use the `unstable` branch and the rest will use the `stable` branch.

Additionally, we maintain separate sets of repositories, branched from `live`, for different teams or projects that require different patching policies appropriate to the needs of those teams or projects. Pulp has strong built-in ACLs that support these divisions.

In order to organize multiple tiers across multiple groups, we use a strict convention to specify the repository ID, which acts as the primary key across all repositories[6], namely:

```
<team name>-<tier>-<os name>-<os version>-
    <arch>-<repo name>
```

For example, `infra-unstable-centos-6-x86_64-updates` would denote the Infrastructure team's `unstable` tier of the 64-bit CentOS 6 "updates" repository. This allows us to tell at a glance the parent-child relationships between repositories.

Sync Filters

The syncs between the `live` and `unstable` and between `unstable` and `stable` tiers are mediated by filters[7]. Filters are regular expression lists of packages to either blacklist from the sync, or whitelist in the sync; in our workflow, only blacklists are used. A package filtered from the sync may still remain in the

repository; that is, if we specify `^kernel(-.*)?` as a blacklist filter, that does not remove `kernel` packages from the repository, but rather refuses to sync new `kernel` packages from the repository's parent. This is critical to our version-pegging system.

Given our needs, whitelist filters are unnecessary; our systems tend to fall into one of two types:

- Systems where we generally want updates to be installed insofar as is reasonable, with some prudence about installing updates to "impactful" packages.
- Systems where, due to vendor requirements, we must set all packages to a specific version. Most often this is in the form of a requirement for a minor release of RHEL[8], in which case there are no updates we wish to install on an automatic basis. (We may wish to update specific packages to respond to security threats, but that happens with manual package promotion, not with a sync; this workflow gives us the flexibility necessary to do so.)

A package that may potentially cause issues when updated can be blacklisted on a per-team basis[9]. Since the repositories are hierarchically tiered, a package that is blacklisted from the `unstable` tier will never make it to the `stable` tier.

Manual Package Promotion and Removal

The lynchpin of this process is manually reviewing packages that have been blacklisted from the syncs and *promoting* them manually as necessary. For instance, if a filter for a set of repositories blacklisted `^kernel(-.*)?` from the sync, without manually promoting new kernel packages no new kernel would ever be installed.

To accomplish this, we use Pulp's *add package* functionality, exposed via the REST API as a POST to `/repositories/<id>/add_package/`, via the Python client API as `pulp.client.api.repository.RepositoryAPI.add_package()`, and via the CLI as `pulp-admin repo add_package`. In the CLI implementation, `add_package` follows dependencies, so promoting a package will promote everything that package requires that is not already in the target repository. This helps ensure that each repository stays consistent even as we manipulate it to contain only a subset of upstream packages[10].

Conversely, if a package is deployed and is later found to cause problems it can be removed from the tier and the previous version, if such is available in the repository, will be (re)installed. Bcfg2 will helpfully flag machines where a newer package is installed than is available in that machine's repositories, and will try to downgrade packages appropriately. Pulp can be configured to retain old packages when it performs a sync; this is helpful for repositories like EPEL that remove old packages themselves, and guarantees that a configurable number of older package versions are available to fall back on.

The *remove package* functionality is exposed via Pulp's REST API as a POST to `/repositories/<id>/delete_package/`, via the Python client API as `pulp.client.api.repository.RepositoryAPI.remove_package()`, and via the CLI as `pulp-admin repo remove_package`. As with `add_package`, the CLI implementation follows dependencies and will try to remove packages that require the package being removed; this also helps ensure repository consistency.

Optimally, security patches are applied 10 or 30 days after the initial patch release [2]; this workflow allows us to follow these recommendations to some degree, promoting new packages to the `unstable` tier on an approximately weekly basis. Packages that have been in the `unstable` tier for at least a week are also promoted to the `stable` tier every week; in this we deviate from Beattie et al.'s recommendations somewhat, but we do so because the updates being promoted to `stable` have been vetted and tested by the machines using the `unstable` tier.

This workflow also gives us something very important: the ability to install updates across all machines much sooner than the optimal 10- or 30-day period. High profile vulnerabilities require immediate action – even to the point of imperiling uptime – and by promoting a new package immediately to both `stable` and `unstable` tiers we can ensure that it is installed across all machines in our environment in a timely fashion.

Selecting "impactful" packages

Throughout this paper, we have referred to "impactful" packages – those to which automatic updates we determined to be particularly dangerous – as a driving factor. Were it not for our reticence to automatically update all packages, we could have simply used an automatic update facility – `yum-cron` or

yum-updatesd are both popular – and been done with it.

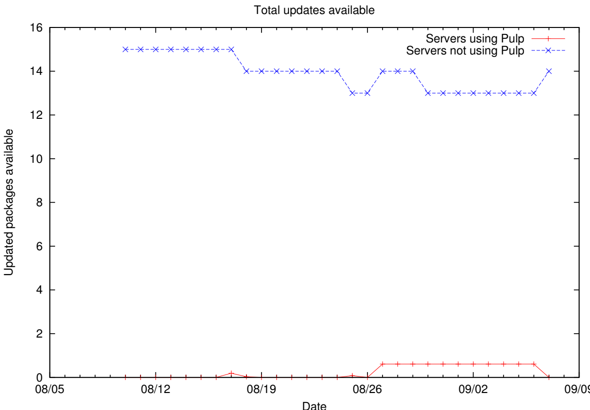
We didn't feel that was appropriate, though. For instance, installing a new kernel can be problematic – particularly in an environment with a wide variety of third-party kernel modules and other kernel-space modifications – and we wanted much closer control over that process. We flagged packages as "impactful" according to a simple set of criteria:

- The kernel, and packages otherwise directly tied to kernel space (e.g., kernel modules and Dynamic Kernel Module Support (DKMS) packages);
- Packages that provide significant, customer-facing services. On the Infrastructure team, this included packages like `bind`, `httpd` (and related modules), `mysql`, and so on.
- Packages related to InfiniBand and Lustre [9]; as one of the world's largest unclassified Lustre installations, it's very important that the Lustre versions on our systems stay in lockstep with all other systems in the center. Parts of Lustre reside directly in kernel space, an additional consideration.

The first two criteria provided around 20 packages to be excluded – a tiny fraction of the total packages installed across all of our machines. The vast majority of supporting packages continue to be automatically updated, albeit with a slight time delay for the multiple syncs that must occur.

Results

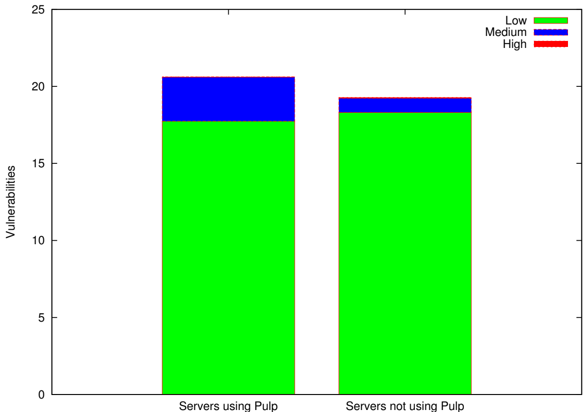
Our approach produces results in a number of areas that are difficult to quantify: improved automation reduces the amount of time we spend installing patches; not installing patches immediately improves patch quality and reduces the likelihood of flawed patches [2]; and increased compartmentalization makes it easier for our diverse teams to work to different purposes without stepping on toes. But it also provides testable, quantifiable improvements: since replacing a manual update process with Pulp and Bcfg2's automated update process, we can see that the number of available updates has decreased and remained low on the machines using Pulp.



The practice of staging package deployment makes is difficult to quantify just how out of date a client is, as yum on the client will only report the number of updates available from the repositories in `yum.conf`. To find the number of updates available from upstream, we collect an aggregate of all the package differences starting at the client and going up the heirarchy to the upstream repository. E.g., for a machine using the `unstable` tier, we calculate the number of updates available on the machine itself, and then the number of updates available to the `unstable` tier from the `live` tier.

The caveat to this approach is when, for instance, a package splits into two new packages. This results in two new packages, and one missing package, totaling three "updates" according to `yum check-update`, or zero "updates" when comparing repositories themselves, when in reality it is a single package update. For example, if package `foo` recieves an update that results in packages `foo-client` and `foo-server`, this could result in a margin of error of -1 or +2. This gives a slight potential benefit to machines using Pulp in our metrics, as updates of this sort are underestimated when calculating the difference between repositories, but overestimated when using `yum` to report on updates available to a machine. In practice, this is extremely rare, though, and should not significantly affect the results.

Ensuring, with a high degree of confidence, that updates are installed is wonderful, but even more important is ensuring that vulnerabilities are being mitigated. Using the data from monthly Nessus [11] vulnerability scans, we can see that machines using Pulp do indeed reap the benefits of being patched with more frequency:[11]



This graph is artificially skewed against Pulp due to the sorts of things Nessus scans for; for instance, web servers are more likely to be using Pulp at this time simply due to our implementation plan, and they also have disproportionately more vulnerabilities in Nessus because they have more services exposed.

Future Development

Sponge

At this time, Pulp is very early code; it has been in use in another Red Hat product for a while, so certain paths are well-tested, but other paths are pre-alpha. Consequently, its command line interface lacks polish, and many tasks within Pulp require extraordinary verbosity to accomplish. It is also not clear if Pulp is intended for standalone use, although such is possible.

To ease management of Pulp, we have written a web frontend for management of Pulp and its objects, called "Sponge." Sponge, powered by the Django [4] web framework, provides views into the state of Pulp repositories along with the ablity to manage its contents. Sponge leverages Pulp's Python client API to provide convience functions that ease our workflow.

By presenting the information visually, Sponge makes repository management much more intuitive. Sponge extends the functionality of Pulp by displaying the differences between a repository and its parent in the form of a diff. These diffs give greater insight into exactly how `stable`, `unstable`, and `live` tiers differ. They also provide insight into the implications of a package promotion or removal.

This is particularly important with package removal, since, as noted, removing a package will also remove anything that requires that specific package. Without Sponge's diff feature and a confirmation step, that is potentially very dangerous; Pulp itself only gives you confirmation of the packages removed without an opportunity to confirm or reject a removal. The contrapositive situation – promoting a package pulling in unintended dependencies – is also potentially dangerous, albeit less so. Sponge helps avert both dangers.

Guaranteeing a minimum package age

As Beattie at al. observe [2], the optimal time to apply security patches is either 10 or 30 days after the patches have been released. Our workflow currently doesn't provide any way to guarantee this; our weekly manual promotion of new packages merely suggests that a patch be somewhere between 0 and 6 days old before it is promoted to `unstable`, and 7 and 13 days old before being promoted to `stable`. We plan to add a feature – either to Sponge or to Pulp – to promote packages only once they have aged properly.

Other packaging formats

In this paper we have dealt with systems using yum and RPM, but the approach can, at least in theory, be expanded to other packaging systems. Pulp intends eventually to support not only Debian packages, but actually any sort of generic content at all [6], making it useful for any packaging system. Bcfg2, for its part, already has package drivers for a wide array of packaging systems, including APT, Solaris packages (Blastwave- or SystemV-style), Encap, FreeBSD packages, IPS, Mac Ports, Pacman, and Portage. This gives a hint of the future potential for this approach.

Availability

Most of the software involved in the approach discussed in this paper is free and open source. The various elements of our solution can be found at:

Pulp `http://pulpproject.org`

Bcfg2 `http://trac.mcs.anl.gov/projects/bcfg2`

Yum `http://yum.baseurl.org/`

Sponge, the web UI to Pulp listed in the Future Development section, is currently incomplete and unreleased. We have already worked closely with the Pulp developers to incorporate features into the Pulp core itself, and we will continue to do so. We hope that Sponge will become unnecessary as Pulp matures.

Author Information

Chris St. Pierre leads the Infrastructure team of the HPC Operations group at the National Center for Computational Sciences at Oak Ridge National Laboratory in Oak Ridge, Tennessee. He is deeply involved with the development of Bcfg2, contributing in particular to the specification validation tool and Packages plugin for the upcoming 1.2.0 release. He has taught widely on internal documentation, LDAP, and spam. Chris serves on the LOPSA Board of Directors.

Matt Hermanson is a member of the Infrastructure team of the HPC Operations group at the National Center for Computational Sciences at Oak Ridge National Laboratory in Oak Ridge, Tennessee. He holds a B.A. in Computer Science from Tennessee Technological University.

References

[1] Anaconda/Kickstart. http://fedoraproject.org/wiki/Anaconda/Kickstart#Chapter_3._Package_Selection.

[2] Beattie, S., Arnold, S., Cowan, C., Wagle, P., Wright, C., and Shostack, A. Timing the Application of Security Patches for Optimal Uptime. Proceedings of LISA '02: Sixteenth Systems Administration Conference, USENIX, pp. 233–42.

[3] Desai, N. Bcfg2. http://trac.mcs.anl.gov/projects/bcfg2.

[4] Django Software Foundation. Django — The Web framework for perfectionists with deadlines. https://www.djangoproject.com/.

[5] Dobies, J. GCRepoApis. https://fedorahosted.org/pulp/wiki/GCRepoApis.

[6] Dobies, J. Generic Content Support. http://blog.pulpproject.org/2011/08/08/generic-content-support/.

[7] Dobies, J. Pulp - Juicy software repository management. http://pulpproject.org.

[8] Jerome, S., Laszlo, T., and St. Pierre, C. Packages. http://docs.bcfg2.org/server/plugins/generators/packages.html.

[9] Oracle Corporation. Lustre. http://wiki.lustre.org/index.php/Main_Page.

[10] Red Hat, Inc. Spacewalk: Free & Open Source Linux Systems Management. http://spacewalk.redhat.com/.

[11] Tenable Network Security. Tenable Nessus. http://www.tenable.com/products/nessus.

[12] Vidal, S. yum. http://yum.baseurl.org/.

[13] Vidal, S. yum.conf - configuration file for yum(8). man 5 yum.conf.

[14] Warner, T., and Sanders, T. The Future of RHN Satellite: A New Architecture Enabling the Traditional Data Center and the Cloud. Red Hat Summit, Red Hat, Inc.

Notes

[1]For instance, Extra Packages for Enterprise Linux (EPEL) and the CentOS repositories themselves.

[2]Admittedly, this is a non-standard naming scheme, but no solution can be predicated on the idea that all RPMs are well-built.

[3]The command in question merely maintains a local file on a machine, so that file would still have to be copied into the Bcfg2 specification, but we believe this would be less error-prone than copying package version details.

[4]This is actually only true if the package is being added from another repository; it is possible to add a package directly from the filesystem, in which case dependency checking is not performed. This is not a use case for us, though.

[5]In Pulp, filters can only be applied to repositories with local feeds.

[6]This may change in future versions of Pulp, as multiple users, ourselves included, have asked for stronger grouping functionality [5].

[7]As noted earlier, in Pulp, filters can only be applied to repositories with local feeds, so no filter mediates the sync between upstream and `live`.

[8]It is lost on many vendors that it is unreasonable and foolish to require a specific RHEL minor release. As much work as has gone into this solution, it is still less than would be required to convince most vendors of this fact, though.

[9]Technically, filters can be applied on a per-repository basis, so black- and whitelists can be applied to individual repositories. This is very rare in our workflow, though.

[10]It is true that our approach does not *guarantee* consistency. A repository sync might result in an inconsistency if a package that was not listed on that sync's blacklist required a package that was listed on the blacklist. In practice this can be limited by using regular expressions to filter families of packages (e.g., `^mysql.*` or `^(.*-)?mysql.*` to blacklist all MySQL-related packages rather than just blacklisting the `mysql-server` package itself

[11]Unfortunately long-term data was not available for vulnerabilities for a number of reasons: CentOS 5 stopped shipping updates in their mainline repositories between July 21st and September 14th; the August security scan was partially skipped; and Pulp hasn't been in production long enough to get meaningful numbers prior to that. Still, the snapshot of data is compelling.

CDE: Run Any Linux Application On-Demand Without Installation

Philip J. Guo
Stanford University
pg@cs.stanford.edu

Abstract

There is a huge ecosystem of free software for Linux, but since each Linux distribution (distro) contains a different set of pre-installed shared libraries, filesystem layout conventions, and other environmental state, it is difficult to create and distribute software that works without hassle across all distros. Online forums and mailing lists are filled with discussions of users' troubles with compiling, installing, and configuring Linux software and their myriad of dependencies. To address this ubiquitous problem, we have created an open-source tool called CDE that automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines. Creating a CDE package is as simple as running the target application under CDE's monitoring, and executing a CDE package requires no installation, configuration, or root permissions. CDE enables Linux users to instantly run any application on-demand without encountering "dependency hell".

1 Introduction

The simple-sounding task of taking software that runs on one person's machine and getting it to run on another machine can be painfully difficult in practice. Since no two machines are identically configured, it is hard for developers to predict the exact versions of software and libraries already installed on potential users' machines and whether those conflict with the requirements of their own software. Thus, software companies devote considerable resources to creating and testing one-click installers for products like Microsoft Office, Adobe Photoshop, and Google Chrome. Similarly, open-source developers must carefully specify the proper dependencies in order to integrate their software into package management systems [4] (e.g., RPM on Linux, MacPorts on Mac OS X). Despite these efforts, online forums and mailing lists are still filled with discussions of users' troubles with compiling, installing, and configuring software and their myriad of dependencies. For example, the official Google Chrome help forum for "install/uninstall issues" has over 5800 threads.

In addition, a study of US labor statistics predicts that by 2012, 13 million American workers will do programming in their jobs, but amongst those, only 3 million will be professional software developers [24]. Thus, there are potentially millions of people who still need to get their software to run on other machines but who are unlikely to invest the effort to create one-click installers or wrestle with package managers, since their primary job is not to release production-quality software. For example:

- **System administrators** often hack together ad-hoc utilities comprised of shell scripts and custom-compiled versions of open-source software, in order to perform system monitoring and maintenance tasks. Sysadmins want to share their custom-built tools with colleagues, quickly deploy them to other machines within their organization, and "future-proof" their scripts so that they can continue functioning even as the OS inevitably gets upgraded.
- **Research scientists** often want to deploy their computational experiments to a cluster for greater performance and parallelism, but they might not have permission from the sysadmin to install the required libraries on the cluster machines. They also want to allow colleagues to run their research code in order to reproduce and extend their experiments.
- **Software prototype designers** often want clients to be able to execute their prototypes without the hassle of installing dependencies, in order to receive continual feedback throughout the design process.

In this paper, we present an open-source tool called CDE [1] that makes it easy for people of all levels of IT expertise to get their software running on other machines without the hassle of manually creating a robust

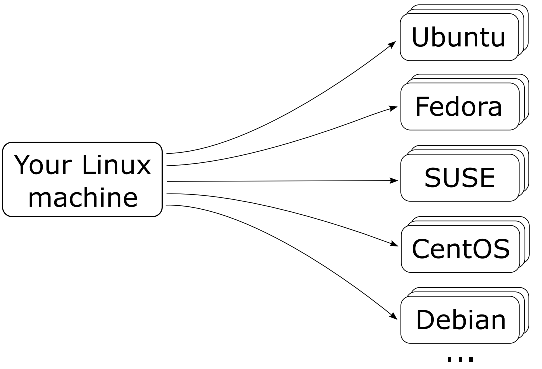


Figure 1: CDE enables users to package up any Linux application and deploy it to all modern Linux distros.

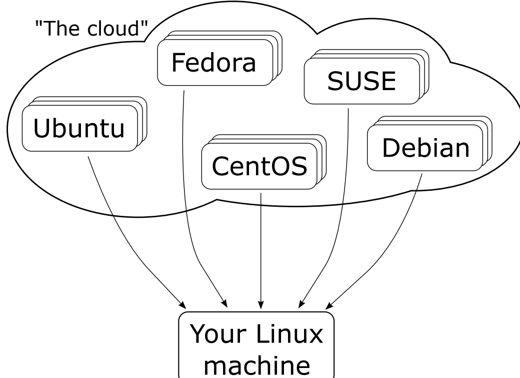


Figure 2: CDE's streaming mode enables users to run any Linux application on-demand by fetching the required files from a farm of pre-installed distros in the cloud.

installer or dealing with user complaints about dependencies. CDE automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines without any installation (see Figure 1). To use CDE, the user simply:

1. Prepends any set of Linux commands with the `cde` executable. `cde` executes the commands and uses `ptrace` system call interposition to collect all the code, data files, and environment variables used during execution into a self-contained package.
2. Copies the resulting CDE package to an x86-Linux machine running any distro from the past ∼5 years.
3. Prepends the original packaged commands with the `cde-exec` executable to run them on the target machine. `cde-exec` uses `ptrace` to redirect file-related system calls so that executables can load the required dependencies from within the package. Execution can range from ∼0% to ∼30% slower.

The main benefits of CDE are that creating a package is as easy as executing the target program under its supervision, and that running a program within a package requires no installation, configuration, or root permissions.

The design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to other Linux machines with as little effort as possible. However, CDE is not meant to replace traditional installers or package managers; its intended role is to serve as a convenient *ad-hoc* solution for people like sysadmins, research scientists, and prototype makers.

Since its release in Nov. 2010, CDE has been downloaded over 3,000 times [1]. We have exchanged hundreds of emails with users throughout both academia and industry. In the past year, we have made several significant enhancements to the base CDE system in response to user feedback. Although we introduced an early version of CDE in a short paper [20], this paper presents a more complete CDE system with three new features:

- To overcome CDE's primary limitation of only being able to package dependencies collected on executed paths, we introduce new tools and heuristics for making CDE packages complete (Section 3).
- To make CDE-packaged programs behave just like native applications on the target machine rather than executing in an isolated sandbox, we introduce a new *seamless execution mode* (Section 4).
- Finally, to enable users to run any Linux application on-demand, we introduce a new *application streaming mode* (Section 5). Figure 2 shows its high-level architecture: The system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). The user connects to that distro farm via an ssh-based protocol from any x86-Linux machine. The user can now run *any* application available within the package managers of any of the distros in the farm. CDE's streaming mode fetches the required files on-demand, caches them locally on the user's machine, and creates a portable distro-independent execution environment. Thus, Linux users can instantly run the hundreds of thousands of applications already available in the package managers of all distros without being forced to use one specific release of one specific distro[1].

This paper continues with descriptions of real-world use cases (Section 6), evaluations of portability and performance (Section 7), comparisons to related work (Section 8), and concludes with discussions of design philosophy, limitations, and lessons learned (Section 9).

[1]The package managers included in different releases of the same Linux distro often contain incompatible versions of many applications!

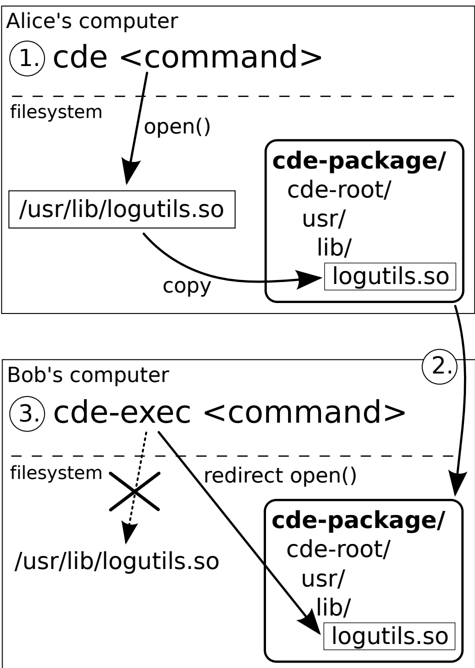


Figure 3: Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends her package to Bob's computer, 3.) Bob runs command with `cde-exec`, which redirects file accesses into package.

2 CDE system overview

We described the details of CDE's design and implementation in a prior paper and its accompanying technical report [20]. We will now summarize the core features of CDE using an example.

Suppose that Alice is a system administrator who is developing a Python script to detect anomalies in network log files. She normally runs her script using this Linux command:

```
python detect_anomalies.py net.log
```

Suppose that Alice's script (`detect_anomalies.py`) imports some 3rd-party Python extension modules, which consist of optimized C++ log parsing code compiled into shared libraries. If Alice wants her colleague Bob to be able to run her analysis, then it is not sufficient to just send her script and `net.log` data file to him.

Even if Bob has a compatible version of Python on his Linux machine, he will not be able to run her script until he compiles, installs, and configures the exact extension modules that her script used (and all of their transitive dependencies). Since Bob is probably using a different Linux distribution (distro) than Alice, even if Alice precisely recalled all of the steps involved in installing all of the original dependencies on her machine, those instructions probably will not work on Bob's machine.

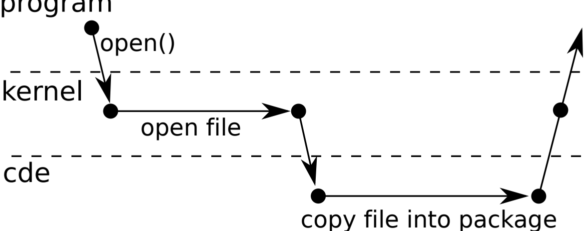


Figure 4: Timeline of control flow between target program, kernel, and `cde` process during an `open` syscall.

2.1 Creating a new CDE package

To create a self-contained package with all of the dependencies required to run her anomaly detection script on another Linux machine, Alice simply prepends her command with the `cde` executable:

```
cde python detect_anomalies.py net.log
```

`cde` runs her command normally and uses the Linux `ptrace` system call to monitor all of the files it accesses throughout execution. `cde` creates a new sub-directory called `cde-package/cde-root/` and copies all of those accessed files into there, mirroring the original directory structure. Figure 4 shows an overview of the control flow between the target program, Linux kernel, and `cde` during a file-related system call.

For example, if Alice's script dynamically loads an extension module as a shared library named `/usr/lib/logutils.so` (i.e., log parsing utility code), then `cde` will copy it to `cde-package/cde-root/usr/lib/logutils.so` (see Figure 3). `cde` also saves the values of environment variables in a text file within `cde-package/`.

When execution terminates, the `cde-package/` sub-directory (which we call a "CDE package") contains all of the files required to run Alice's original command.

2.2 Executing a CDE package

Alice zips up the `cde-package/` directory and transfers it to Bob's Linux machine. Now Bob can run Alice's anomaly detection script without first installing anything on his machine. To do so, he unzips the package, changes into the sub-directory containing the script, and prepends her original command with the `cde-exec` executable (also included in the package):

```
cde-exec python detect_anomalies.py net.log
```

`cde-exec` sets up the environment variables saved from Alice's machine and executes the versions of `python` and its extension modules that are *located within the package*. `cde-exec` uses `ptrace` to monitor all

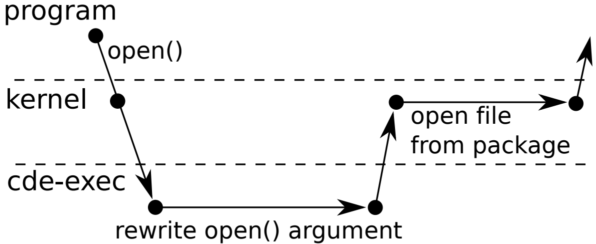


Figure 5: Timeline of control flow between target program, kernel, and `cde-exec` during an `open` syscall.

system calls that access files and dynamically rewrites their path arguments to the corresponding paths within the `cde-package/cde-root/` sub-directory. Figure 5 shows the control flow between the target program, kernel, and `cde-exec` during a file-related system call.

For example, when her script requests to load the `/usr/lib/logutils.so` library using an `open` system call, `cde-exec` rewrites the path argument of the `open` call to `cde-package/cde-root/usr/lib/logutils.so` (see Figure 3). This run-time path redirection is essential, because `/usr/lib/logutils.so` probably does not exist on Bob's machine.

2.3 CDE package portability

Alice's CDE package can execute on any Linux machine with an architecture and kernel version that are compatible with its constituent binaries. CDE currently works on 32-bit and 64-bit variants of the x86 architecture (i386 and x86-64, respectively). In general, a 32-bit `cde-exec` can execute 32-bit packaged applications on 32- and 64-bit machines. A 64-bit `cde-exec` can execute both 32-bit and 64-bit packaged applications on a 64-bit machine. Extending CDE to other architectures (e.g., ARM) is straightforward because the `strace` tool that CDE is built upon already works on many architectures. However, CDE packages cannot be transported *across* architectures without using a CPU emulator.

Our portability experiments (§7.1) show that packages are portable across Linux distros released within 5 years of the distro where the package originated. Besides sharing with colleagues like Bob, Alice can also deploy her package to run on a cluster for more computational power or to a public-facing server machine for real-time online monitoring. Since she does not need to install anything as root, she does not risk perturbing existing software on those machines. Also, having her script and all of its dependencies (including the Python interpreter and extension modules) encapsulated within a CDE package makes it somewhat "future-proof" and likely to continue working on her machine even when its version of Python and associated extensions are upgraded in the future.

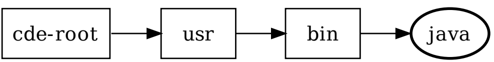


Figure 6: The result of copying a file named `/usr/bin/java` into the `cde-root/` directory.

3 Semi-automated package completion

CDE's primary limitation is that it can only package up files accessed on executed program paths. Thus, programs run from within a CDE package will fail when executing paths that access new files (e.g., libraries, configuration files) that the original execution(s) did not access.

Unfortunately, *no automatic tool* (static or dynamic) can find and package up all the files required to successfully execute all possible program paths, since that problem is undecidable in general. Similarly, it is also impossible to automatically quantify how "complete" a CDE package is or determine what files are missing, since every file-related system call instruction could be invoked with complex or non-deterministic arguments. For example, the Python interpreter executable has only one `dlopen` call site for dynamically loading extension modules, but that `dlopen` could be called many times with different dynamically-generated string arguments derived from script variables or configuration files.

There are two ways to cope with this package incompleteness problem. First, if the user executes additional program paths, then CDE will add new files into the same `cde-package/` directory. However, making repeated executions can get tedious, and it is unclear how many or which paths are necessary to complete the package[2].

Another way to make CDE packages more complete is by manually copying additional files and subdirectories into `cde-package/cde-root/`. For example, while executing a Python script, CDE might automatically copy the few Python standard library files it accesses into, say, `cde-package/cde-root/usr/lib/python/`. To complete the package, the user could copy the entire `/usr/lib/python/` directory into `cde-package/cde-root/` so that *all* Python libraries are present. A user can usually make his/her package complete by copying only a few crucial directories into the package, since programs store all of their files in several top-level directories (see Section 3.3).

However, programs also depend on shared libraries that reside in system-wide directories like `/lib` and `/usr/lib`. Copying all the contents of those directories into a package results in lots of wasted disk space. In Section 3.2, we present an automatic heuristic technique that finds nearly all shared libraries that a program requires and copies them into the package.

[2] similar to trying to achieve 100% coverage during software testing

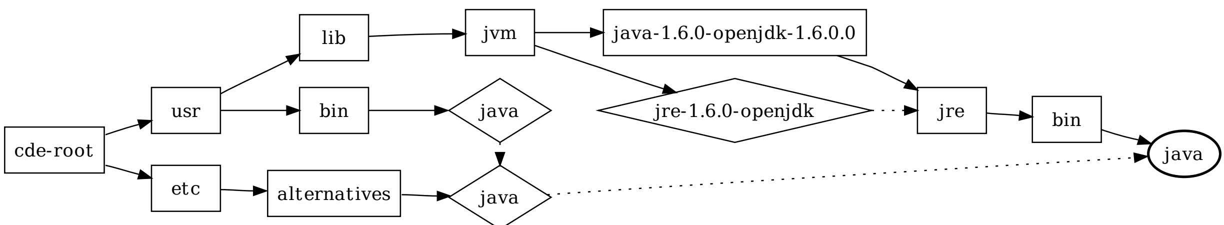


Figure 7: The result of using OKAPI to deep-copy a single `/usr/bin/java` file into `cde-root/`, preserving the exact symlink structure from the original directory tree. Boxes are directories (solid arrows point to their contents), diamonds are symlinks (dashed arrows point to their targets), and the bold ellipse is the actual `java` executable file.

3.1 The OKAPI utility for deep file copying

Before describing our heuristics for completing CDE packages, we first introduce a utility library we built called OKAPI (pronounced "*oh-copy*"), which performs detailed copying of files, directories, and symlinks. OKAPI does one seemingly-simple task that turns out to be tricky in practice: copying a filesystem entity (i.e., a file, directory, or symlink) from one directory to another while fully preserving its original sub-directory and symlink structure (a process that we call *deep-copying*). CDE uses OKAPI to copy files into the `cde-root/` sub-directory when creating a new package, and the support scripts of Sections 3.2 and 3.3 also use OKAPI.

For example, suppose that CDE needs to copy the `/usr/bin/java` executable file into `cde-root/` when it is packaging a Java application. The straightforward way to do this is to use the standard `mkdir` and `cp` utilities. Figure 6 shows the resulting sub-directory structure within `cde-root/`, with the boxes representing directories and the bold ellipse representing the copy of the `java` executable file located at `cde-root/usr/bin/java`. However, it turns out that if CDE were to use this straightforward copying method, the Java application would *fail to run* from within the CDE package! This failure occurs because the `java` executable introspects its own path and uses it as the search path for finding the Java standard libraries. On our Fedora Core 9 machine, the Java standard libraries are actually installed in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0`, so when `java` reads its own path as `/usr/bin/java`, it cannot possibly use that path to find its standard libraries.

In order for Java applications to properly run from within CDE packages, all of their constituent files must be "deep-copied" into the package while replicating their original sub-directory and symlink structures. Figure 7 illustrates the complexity of deep-copying a single file, `/usr/bin/java`, into `cde-root/`. The diamond-shaped nodes represent symlinks, and the dashed arrows point to their targets. Notice how `/usr/bin/java` is a symlink to `/etc/alternatives/java`, which is itself a symlink to `/usr/lib/jvm/jre-1.6.0-openjdk/bin/java`. Another complicating factor is that `/usr/lib/jvm/jre-1.6.0-openjdk` is itself a symlink to the `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/` directory, so the actual `java` executable resides in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/bin/`. Java can only find its standard libraries when these paths are all faithfully replicated within the CDE package.

The OKAPI utility library automatically performs the deep-copying required to generate the filesystem structure of Figure 7. Its interface is as simple as ordinary `cp`: The caller simply requests for a path to be copied into a target directory, and OKAPI faithfully replicates the sub-directory and symlink structure.

OKAPI performs one additional task: rewriting the contents of symlinks to transform absolute path targets into relative path targets within the destination directory (e.g., `cde-root/`). In our example, `/usr/bin/java` is a symlink to `/etc/alternatives/java`. However, OKAPI cannot simply create the `cde-root/usr/bin/java` symlink to also point to `/etc/alternatives/java`, since that target path is outside of `cde-root/`. Instead, OKAPI must rewrite the symlink target so that it actually refers to `../../etc/alternatives/java`, which is a relative path that points to `cde-root/etc/alternatives/java`.

The details of this particular example are not important, but the high-level message that Figure 7 conveys is that deep-copying even a single file can lead to the creation of over a dozen sub-directories and (possibly-rewritten) symlinks. The problem that OKAPI solves is not Java-specific; we have observed that many real-world Linux applications fail to run from within CDE packages unless their files are deep-copied in this detailed way.

OKAPI is also available as a free standalone command-line tool [1]. To our knowledge, no other Linux file copying tool (e.g., `cp`, `rsync`) can perform the deep-copying and symlink rewriting that OKAPI does.

3.2 Heuristics for copying shared libraries

When Linux starts executing a dynamically-linked executable, the dynamic linker (e.g., `ld-linux*.so*`) finds and loads all shared libraries that are listed in a special `.dynamic` section within the executable file. Running the `ldd` command on the executable shows these start-up library dependencies. When CDE is executing a target program to create a package, CDE finds all of these dependencies as well because they are loaded at start-up time via `open` system calls.

However, programs sometimes load shared libraries in the middle of execution using, say, the `dlopen` function. This run-time loading occurs mostly in GUI programs with a plug-in or extension architecture. For example, when the user instructs Firefox to visit a web page with a Flash animation, Firefox will use `dlopen` to load the Adobe Flash Player shared library. `ldd` will not find that dependency since it is not hard-coded in the `.dynamic` section of the Firefox executable, and CDE will only find that dependency if the user actually visits a Flash-enabled web page while creating a package for Firefox.

We have created a simple heuristic-based script that finds most or all shared libraries that a program requires[3]. The user first creates a base CDE package by executing the target program once (or a few times) and then runs our script, which works as follows:

1. Find all ELF binaries (executables and shared libraries) within the package using the Linux `find` and `file` utilities.
2. For each binary, find all constant strings using the `strings` utility, and look for strings containing "`.so`" since those are likely to be shared libraries.
3. Call the `locate` utility on each candidate shared library string, which returns the *full absolute paths* of all installed shared libraries that match each string.
4. Use OKAPI to copy each library into the package.
5. Repeat this process until no new libraries are found.

This heuristic technique works well in practice because programs often list all of their dependent shared libraries in string *constants* within their binaries. The main exception occurs in dynamic languages like Python or MATLAB, whose programs often dynamically generate shared library paths based on the contents of scripts and configuration files.

Another limitation of this technique is that it is overly conservative and can create larger-than-needed packages, since the `locate` utility can find more libraries than the target program actually needs.

[3] always a superset of the shared libraries that `ldd` finds

3.3 OKAPI-based directory copying script

In general, running an application once under CDE monitoring only packages up a subset of all required files. In our experience, the easiest way to make CDE packages complete is to copy entire sub-directories into the package. To facilitate this process, we created a script that repeatedly calls OKAPI to copy an entire directory at a time into `cde-root/`, automatically following symlinks to other directories and recursively copying as needed.

Although this approach might seem primitive, it is effective in practice because applications often store all of their files in a few top-level directories. When a user inspects the directory structure within `cde-root/`, it is usually obvious where the application's files reside. Thus, the user can run our OKAPI-based script to copy the entirety of those directories into the package.

Evaluation: To demonstrate the efficacy of this approach, we have created complete self-contained CDE packages for six of the largest and most popular Linux applications. For each app, we made an initial packaging run with `cde`, inspected the package contents, and copied at most three directories into the package. The entire packaging process took several minutes of human effort per application. Here are our full results:

- **AbiWord** is a free alternative to Microsoft Word. After an initial packaging run, we saw that some plug-ins were included in the `cde-root/usr/lib/abiword-2.8/plugins` and `cde-root/usr/lib/goffice/0.8.1/plugins` directories. Thus, we copied the entirety of those two original directories into `cde-root/` to complete its package, thereby including all AbiWord plug-ins.
- **Eclipse** is a sophisticated IDE and software development platform. We completed its package by copying the `/usr/lib/eclipse` and `/usr/share/eclipse` directories into `cde-root/`.
- **Firefox** is a popular web browser. We completed its package by copying `/usr/lib/firefox-3.6.18` and `/usr/lib/firefox-addons` into `cde-root/` (plus another directory for the third-party Adobe Flash player plug-in).
- **GIMP** is a sophisticated graphics editing tool. We completed its package by copying `/usr/lib/gimp/2.0` and `/usr/share/gimp/2.0`.
- **Google Earth** is an interactive 3D mapping application. We completed its package by copying `/opt/google/earth` into `cde-root/`.
- **OpenOffice.org** is a free alternative to the Microsoft Office productivity suite. We completed its package by copying the `/usr/lib/openoffice` directory into `cde-root/`.

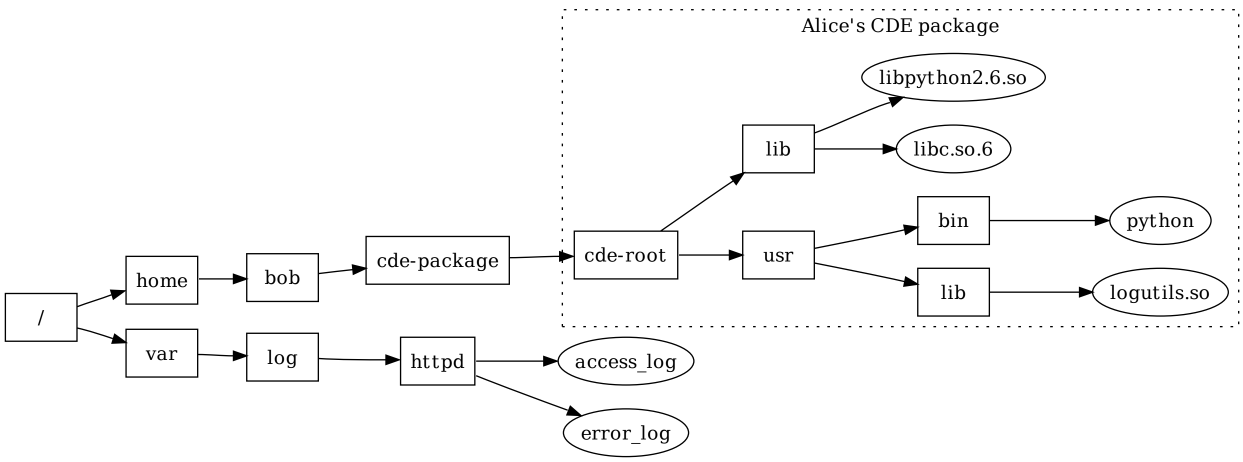


Figure 8: Example filesystem layout on Bob's machine after he receives a CDE package from Alice (boxes are directories, ellipses are files). CDE's seamless execution mode enables Bob to run Alice's packaged script on the log files in `/var/log/httpd/` without first moving those files inside of `cde-root/`.

4 Seamless execution mode

When executing a program from within a package, `cde-exec` redirects all file accesses into the package by default, thereby creating a chroot-like sandbox with `cde-package/cde-root/` as the pseudo-root directory (see Figure 3, Step 3). However, unlike chroot, CDE does not require root access to run, and its sandbox policies are flexible and user-customizable [20].

This default chroot-like execution mode is fine for running self-contained GUI applications like games or web browsers, but it is a somewhat awkward way to run most types of UNIX-style command-line programs that system administrators, developers, and hackers often prefer. If users are running, say, a compiler or command-line image processing utility from within a CDE package, they would need to first move their input data files into the package, run the target program using `cde-exec`, and then move the resulting output data files back out of the package, which is a cumbersome process.

In our Alice-and-Bob example from Section 2 (see Figure 3), if Bob wants to run Alice's anomaly detection script on his own log data (e.g., `bob.log`), he needs to first move his data file inside of `cde-package/cde-root/`, change into the appropriate sub-directory deep within the package, and then run:

```
cde-exec python detect_anomalies.py bob.log
```

In contrast, if Bob had actually installed the proper version of Python and its required extension modules on his machine, then he could run Alice's script from *anywhere* on his filesystem with no restrictions. Some CDE users wanted CDE-packaged programs to behave just like regularly-installed programs rather than requiring input files to be moved inside of a `cde-package/cde-root/` sandbox, so we implemented a new *seamless execution mode* that largely achieves this goal.

Seamless execution mode works using a simple heuristic: If `cde-exec` is being invoked from a directory *not* in the CDE package (i.e., from somewhere else on the user's filesystem), then only redirect a path into `cde-package/cde-root/` if the file that the path refers to actually exists within the package. Otherwise simply leave the path unmodified so that the program can access the file normally. No user intervention is needed in the common case.

The intuition behind why this heuristic works is that when programs request to load libraries and other mandatory components, those files must exist within the package, so their paths are redirected. On the other hand, when programs request to load an input file passed via, say, a command-line argument, that file does not exist within the package, so the original path is used to retrieve it from the native filesystem.

In the example shown in Figure 8, if Bob ran Alice's script to analyze an arbitrary log file on his machine (e.g., his web server log, `/var/log/httpd/access_log`), then `cde-exec` will redirect Python's request for its own libraries (e.g., `/lib/libpython2.6.so` and `/usr/lib/logutils.so`) inside of `cde-root/` since those files exist within the package, but `cde-exec` will *not* redirect `/var/log/httpd/access_log` and instead load the real file from its original location.

Seamless execution mode fails when the user wants the packaged program to access a file from the native filesystem, but an identically-named file actually exists within the package. In the above example, if `cde-package/cde-root/var/`

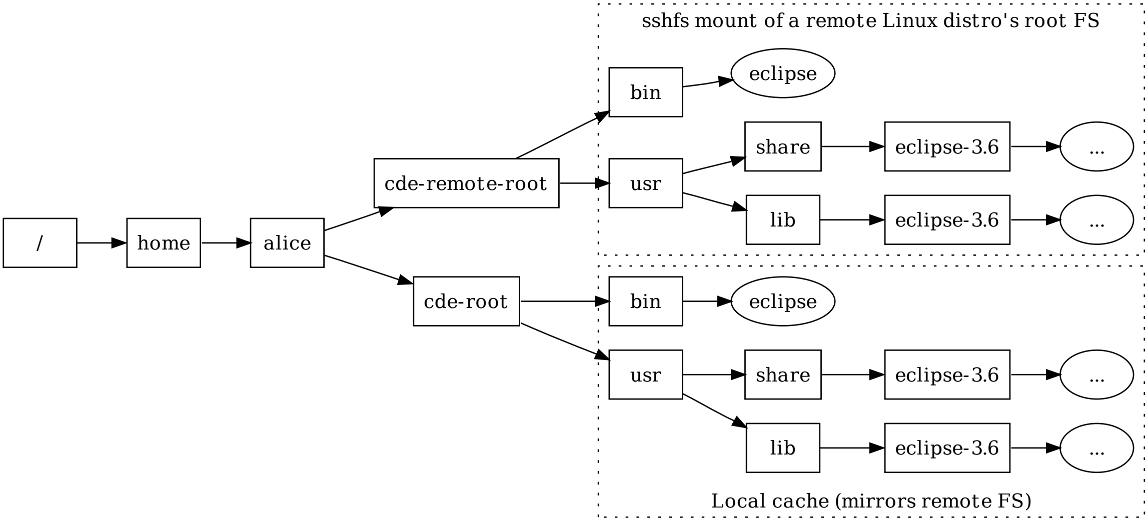


Figure 9: An example use of CDE's streaming mode to run Eclipse 3.6 on any Linux machine without installation. `cde-exec` fetches all dependencies on-demand from a remote Linux distro and stores them in a local cache.

`log/httpd/access_log` existed, then that file would be processed by the Python script instead of `/var/log/httpd/access_log`. There is no automated way to resolve such name conflicts, but `cde-exec` provides a "verbose mode" where it prints out a log of what paths were redirected within the package. The user can inspect that log and then manually write redirection/ignore rules in a configuration file to control which paths `cde-exec` redirects into `cde-root/`. For instance, the user could tell `cde-exec` to *not* redirect any paths starting with `/var/log/httpd/*`.

Using seamless execution mode, our users have been able to run software such as programming language interpreters and compilers, scientific research tools, and sysadmin scripts from CDE packages and have them behave just like regularly-installed programs.

5 On-demand application streaming

We now introduce a new application streaming mode where CDE users can instantly run any Linux application on-demand without having to create, transfer, or install any packages. Figure 2 shows a high-level architectural overview. The basic idea is that a system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). When a user wants to run some application that is available on a particular distro, they use sshfs (an ssh-based network filesystem [9]) to mount the root directory of that distro into a special `cde-remote-root/` mountpoint on their Linux machine. Then the user can use CDE's streaming mode to run any application from that distro locally on their own machine.

5.1 Implementation and example

Figure 9 shows an example of streaming mode. Let's say that Alice wants to run the Eclipse 3.6 IDE on her Linux machine, but the particular distro she is using makes it difficult to obtain all the dependencies required to install Eclipse 3.6. Rather than suffering through dependency hell, Alice can simply connect to a distro in the farm that contains Eclipse 3.6 and then use CDE's streaming mode to "harvest" the required dependencies on-demand.

Alice first mounts the root directory of the remote distro at `cde-remote-root/`. Then she runs "`cde-exec -s eclipse`" (`-s` activates streaming mode). `cde-exec` finds and executes `cde-remote-root/bin/eclipse`. When that executable requests shared libraries, plug-ins, or any other files, `cde-exec` will redirect the respective paths into `cde-remote-root/`, thereby executing the version of Eclipse 3.6 that resides in the cloud distro. However, note that the application is running locally on Alice's machine, not in the cloud.

An astute reader will immediately realize that running applications in this manner can be slow, since files are being accessed from a remote server. While sshfs performs some caching, we have found that it does not work well enough in practice. Thus, we have implemented our own caching layer within CDE: When a remote file is accessed from `cde-remote-root/`, `cde-exec` uses OKAPI to make a deep-copy into a local `cde-root/` directory and then redirects that file's path into `cde-root/`. In streaming mode, `cde-root/` initially starts out empty and then fills up with a subset of files from `cde-remote-root/` that the target program has accessed.

To avoid unnecessary filesystem accesses, CDE's cache also keeps a list of file paths that the target program tried to access from the remote server, even keeping paths for *non-existent files*. On subsequent runs, when the program tries to access one of those paths, `cde-exec` will redirect the path into the local `cde-root/` cache. It is vital to track non-existent files since programs often try to access non-existent files at start-up while doing, say, a search for shared libraries by probing a list of directories in a search path. If CDE did not track non-existent files, then the program would still access the directory entries on the remote server before discovering that those files still do not exist, thus slowing down performance.

With this cache in place, the first time an application is run, all of its dependencies must be downloaded, which could take several seconds to minutes. This one-time delay is unavoidable. However, subsequent runs simply use the files already in the local cache, so they execute at regular `cde-exec` speeds. An added bonus is that even running a *different* application for the first time might still result in some cache hits for, say, generic libraries like `libc`, so the entire application does not need to be downloaded.

Finally, the package incompleteness problem faced by regular CDE (see Section 3) no longer exists in streaming mode. When the target application needs to access new files that do not yet exist in the local cache (e.g., Alice loads a new Eclipse plug-in), those files are transparently fetched from the remote server and cached.

5.2 Synergy with package managers

Nearly all Linux users are currently running one particular distro with one default package manager that they use to install software. For instance, Ubuntu users must use APT, Fedora users must use YUM, SUSE users must use Zypper, Gentoo users must use Portage, etc. Moreover, different releases of the *same* distro contain different software package versions, since distro maintainers add, upgrade, and delete packages in each new release[4].

As long as a piece of software and all of its dependencies are present within the package manager of the exact distro release that a user happens to be using, then installation is trivial. However, as soon as even one dependency cannot be found within the package manager, then users must revert to the arduous task of compiling from source (or configuring a custom package manager).

CDE's streaming mode frees Linux users from this single-distro restriction and allows them to run software that is available within the package manager of any distro in the cloud distro farm. The system administrator is responsible for setting up the farm and provisioning access rights (e.g., ssh keys) to users. Then users can directly install packages in any cloud distro and stream the desired applications to run locally on their own machines.

Philosophically, CDE's streaming mode maximizes user freedom since users are now free to run any application in any package manager from the comfort of their own machines, regardless of which distro they choose to use. CDE complements traditional package managers by leveraging all of the work that the maintainers of each distro have already done and opening up access to users of all other distros. This synergy can potentially eliminate quasi-religious squabbles and flame-wars over the virtues of competing distros or package management systems. Such fighting is unnecessary since CDE allows users to freely choose from amongst all of them.

6 Real-world use cases

Since we released the first version of CDE on November 9, 2010, it has been downloaded at least 3,000 times as of September 2011 [1]. We cannot track how many people have directly checked out its source code from GitHub, though. We have exchanged hundreds of emails with CDE users and discovered six salient real-world use cases as a result of these discussions. Table 1 shows that we used 16 CDE packages, mostly sent in by our users, as benchmarks in the experiments reported in Section 7. They contain software written in diverse programming languages and frameworks. We now summarize the use case categories and benchmarks (highlighted in **bold**).

Distributing research software: The creators of two research tools found CDE online and used it to create portable packages that they uploaded to their websites:

The website for **`graph-tool`**, a Python/C++ module for analyzing graphs, lists these (direct) dependencies: "GCC 4.2 or above, Boost libraries, Python 2.5 or above, expat library, NumPy and SciPy Python modules, GCAL C++ geometry library, and Graphviz with Python bindings enabled." [11] Unsurprisingly, lots of people had trouble compiling it: 47% of all messages on its mailing list (137 out of 289) were questions related to compilation problems. The author of `graph-tool` used CDE to automatically create a portable package (containing 149 shared libraries and 1909 total files) and uploaded it to his website so that users no longer needed to suffer through the pain of manually compiling it.

`arachni`, a Ruby-based tool that audits web application security [10], requires six hard-to-compile Ruby extension modules, some of which depend on versions of Ruby and libraries that are not available in the pack-

[4] We once tried installing a machine learning application that depended on the `libcv` computer vision library. The required `libcv` version was found in the APT repository on Ubuntu 10.04, but it was not found in the repositories on the two immediately neighboring Ubuntu releases: 9.10 and 10.10.

Package name	Description	Dependencies	Creator
Distributing research software			
arachni	Web app. security scanner framework [10]	Ruby (+ extensions)	security researcher
graph-tool	Lib. for manipulation & analysis of graphs [11]	Python, C++, Boost	math researcher
pads	Language for processing ad-hoc data [19]	Perl, ML, Lex, Yacc	self
saturn	Static program analysis framework [13]	Perl, ML, Berkeley DB	self
Running production software on incompatible distros			
meld	Interactive visual diff and merge tool for text	Python, GTK+	software engineer
bio-menace	Classic video game within a MS-DOS emulator	DOSBox, SDL	game enthusiast
google-earth	3D interactive map application by Google	shell scripts, OpenGL	self
Creating reproducible computational experiments			
kpiece	Robot motion planning algorithm [26]	C++, OpenGL	robotics researcher
gadm	Genetic algorithm for social networks [21]	C++, make, R	self
Deploying computations to cluster or cloud			
ztopo	Batch processing of topological map images	C++, Qt	graduate student
klee	Automatic bug finder & test case generator [16]	C++, LLVM, μClibc	self
Submitting executable bug reports			
coq-bug-2443	Incorrect output by Coq proof assistant [2]	ML, Coq	bug reporter
gcc-bug-46651	Causes GCC compiler to segfault [3]	gcc	bug reporter
llvm-bug-8679	Runs LLVM compiler out of memory [5]	C++, LLVM	bug reporter
Collaborating on class programming projects			
email-search	Natural language semantic email search	Python, NLTK, Octave	college student
vr-osg	3D virtual reality modeling of home appliances	C++, OpenSceneGraph	college student

Table 1: CDE packages used as benchmarks in our experiments, grouped by use cases. 'self' in the 'Creator' column means package was created by the author; all other packages created by CDE users (mostly people we have never met).

age managers of most modern Linux distributions. Its creator, a security researcher, created and uploaded CDE packages and then sent us a grateful email describing how much effort CDE saved him: *"My guess is that it would take me half the time of the development process to create a self-contained package by hand; which would be an unacceptable and truly scary scenario."*

In addition, we used CDE to create portable binary packages for two of our Stanford colleagues' research tools, which were originally distributed as tarballs of source code: **pads** [19] and **saturn** [13]. 44% of the messages on the pads mailing list (38 / 87) were questions related to troubles with compiling it (22% for saturn). Once we successfully compiled these projects (after a few hours of improvising our own hacks since the instructions were outdated), we created CDE packages by running their regression test suites, so that others do not need to suffer through the compilation process.

Even the saturn team leader admitted in a public email, *"As it stands the current release likely has problems running on newer systems because of bit rot — some libraries and interfaces have evolved over the past couple of years in ways incompatible with the release."* [7] In contrast, our CDE packages are largely immune to "bit rot" (until the user-kernel ABI changes) because they contain all required dependencies.

Running software on incompatible distros: Even production-quality software might be hard to install on Linux distros with older kernel or library versions, especially when system upgrades are infeasible. For example, an engineer at Cisco wanted to run some new open-source tools on his work machines, but the IT department mandated that those machines run an older, more secure enterprise Linux distro. He could not install the tools on those machines because that older distro did not have up-to-date libraries, and he was not allowed to upgrade. Therefore, he installed a modern distro at home, ran CDE on there to create packages for the tools he wanted to port, and then ran the tools from within the packages on his work machines. He sent us one of the packages, which we used as a benchmark: the **meld** visual diff tool.

Hobbyists applied CDE in a similar way: A game enthusiast could only run a classic game (**bio-menace**) within a DOS emulator on one of his Linux machines, so he used CDE to create a package and can now play the game on his other machines. We also helped a user create a portable package for the Google Earth 3D map application (**google-earth**), so he can now run it on older distros whose libraries are incompatible with Google Earth.

Reproducible computational experiments: A fundamental tenet of science is that colleagues should be able to reproduce the results of one's experiments. In the past few years, science journals and CS conferences (e.g., SIGMOD, FSE) have encouraged authors of published papers to put their code and datasets online, so that others can independently re-run, verify, and build upon their experiments. However, it can be hard for people to set up all of the (often-undocumented) dependencies required to re-run experiments. In fact, it can even be difficult to re-run *one's own experiments* in the future, due to inevitable OS and library upgrades. To ensure that he could later re-run and adjust experiments in response to reviewer critiques for a paper submission [16], our groupmate Cristian took the hard drive out of his computer at paper submission time and archived it in his drawer!

In our experience, the results of many computational science experiments can be reproduced within CDE packages since the programs are output-deterministic [15], always producing the same outputs (e.g., statistics, graphs) for a given input. A robotics researcher used CDE to make the experiments for his motion planning paper (**kpiece**) [26] fully-reproducible. Similarly, we helped a social networking researcher create a reproducible package for his genetic algorithm paper (**gadm**) [21].

Deploying computations to cluster or cloud: People working on computational experiments on their desktop machines often want to run them on a cluster for greater performance and parallelism. However, before they can deploy their computations to a cluster or cloud computing (e.g., Amazon EC2), they must first install all of the required executables and dependent libraries on the cluster machines. At best, this process is tedious and time-consuming; at worst, it can be impossible, since regular users often do not have root access on cluster machines.

A user can create a self-contained package using CDE on their desktop machine and then execute that package on the cluster or cloud (possibly many instances in parallel), without needing to install any dependencies or to get root access on the remote machines. For instance, our colleague Peter wanted to use a department-administered 100-CPU cluster to run a parallel image processing job on topological maps (**ztopo**). However, since he did not have root access on those older machines, it was nearly impossible for him to install all of the dependencies required to run his computation, especially the image processing libraries. Peter used CDE to create a package by running his job on a small dataset on his desktop, transferred the package and the complete dataset to the cluster, and then ran 100 instances of it in parallel there.

Similarly, we worked with lab-mates to use CDE to deploy the CPU-intensive **klee** [16] bug finding tool from the desktop to Amazon's EC2 cloud computing service without needing to compile Klee on the cloud machines. Klee can be hard to compile since it depends on LLVM, which is very picky about specific versions of GCC and other build tools being present before it will compile.

Submitting executable bug reports: Bug reporting is a tedious manual process: Users submit reports by writing down the steps for reproduction, exact versions of executables and dependent libraries, (e.g., *"I'm running Java version 1.6.0_13, Eclipse SDK Version 3.6.1, ..."*), and maybe attaching an input file that triggers the bug. Developers often have trouble reproducing bugs based on these hand-written descriptions and end up closing reports as "not reproducible."

CDE offers an easier and more reliable solution: The bug reporter can simply run the command that triggers the bug under CDE supervision to create a CDE package, send that package to the developer, and the developer can re-run that same command on their machine to reproduce the bug. The developer can also modify the input file and command-line parameters and then re-execute, in order to investigate the bug's root cause.

To show that this technique works, we asked people who recently reported bugs to popular open-source projects to use CDE to create executable bug reports. Three volunteers sent us CDE packages, and we were able to reproduce all of their bugs: one that causes the Coq proof assistant to produce incorrect output (**coq-bug-2443**) [2], one that segfaults the GCC compiler (**gcc-bug-46651**) [3], and one that makes the LLVM compiler allocate an enormous amount of memory and crash (**llvm-bug-8679**) [5].

Since CDE is not a record-replay tool, it is not guaranteed to reproduce non-deterministic bugs. However, at least it allows the developer to run the exact versions of the faulting executables and dependent libraries.

Collaborating on class programming projects: Two users sent us CDE packages they created for collaborating on class assignments. Rahul, a Stanford grad student, was using NLTK [22], a Python module for natural language processing, to build a semantic email search engine (**email-search**) for a machine learning class. Despite much struggle, Rahul's two teammates were unable to install NLTK on their Linux machines due to conflicting library versions and dependency hell. This meant that they could only run one instance of the project at a

time on Rahul's laptop for query testing and debugging. When Rahul discovered CDE, he created a package for their project and was able to run it on his two teammates' machines, so that all three of them could test and debug in parallel. Joshua, an undergrad from Mexico, emailed us a similar story about how he used CDE to collaborate on and demo his virtual reality class project (**vr-osg**).

7 Evaluation

7.1 Evaluating CDE package portability

To show that CDE packages can successfully execute on a wide range of Linux distros and kernel versions, we tested our benchmark packages on popular distros from the past 5 years. We installed fresh copies of these distros (listed with the versions and release dates of their kernels) on a 3GHz Intel Xeon x86-64 machine:

- Sep 2006 — CentOS 5.5 (Linux 2.6.18)
- Oct 2007 — Fedora Core 8 (Linux 2.6.23)
- Oct 2008 — openSUSE 11.1 (Linux 2.6.27)
- Sep 2009 — Ubuntu 9.10 (Linux 2.6.31)
- Feb 2010 — Mandriva Free Spring (Linux 2.6.33)
- Aug 2010 — Linux Mint 10 (Linux 2.6.35)

We installed 32-bit and 64-bit versions of each distro and executed our 32-bit benchmark packages (those created on 32-bit distros) on the 32-bit versions, and our 64-bit packages on the 64-bit versions. Although all of these distros reside on one physical machine, none of our benchmark packages were created on that machine: CDE users created most of the packages, and we made sure to create our own packages on other machines.

Results: Out of the 96 unique configurations we tested (16 CDE packages each run on 6 distros), all executions succeeded except for one[5]. By "succeeded", we mean that the programs ran correctly, as far as we could observe: Batch programs generated identical outputs across distros; regression tests passed; we could interact normally with the GUI programs; and we could reproduce the symptoms of the executable bug reports.

In addition, we were able to successfully execute all of our 32-bit packages on the 64-bit versions of CentOS, Mandriva, and openSUSE (the other 64-bit distros did not support executing 32-bit binaries).

In sum, we were able to use CDE to successfully execute a diverse set of programs (Table 1) "out-of-the-box" on a variety of Linux distributions from the past 5 years, without performing any installation or configuration.

[5] `vr-osg` failed on Fedora Core 8 with a known error related to graphics drivers.

7.2 Comparing against a one-click installer

To show that the level of portability that CDE enables is substantive, we compare CDE against a representative one-click installer for a commercial application. We installed and ran Google Earth (Version 5.2.1, Sep 2010) on our 6 test distros using the official 32-bit installer from Google. Here is what happened on each distro:

- CentOS (Linux 2.6.18) — installs fine but Google Earth crashes upon start-up with variants of this error message repeated several times, because the GNU Standard C++ Library on this OS is too old:

```
/usr/lib/libstdc++.so.6:
version `GLIBCXX_3.4.9' not found
(required by ./libgoogleearth_free.so)
```

- Fedora (Linux 2.6.23) — same error as CentOS
- openSUSE (Linux 2.6.27) — installs and runs fine
- Ubuntu (Linux 2.6.31) — installs and runs fine
- Mandriva (Linux 2.6.33) — installs fine but Google Earth crashes upon start-up with this error message because a required graphics library is missing:

```
error while loading shared libraries:
libGL.so.1: cannot open shared object
file: No such file or directory
```

- Linux Mint (Linux 2.6.35) — installer program crashes with this cryptic error message because the XML processing library on this OS is *too new* and thus incompatible with the installer:

```
setup.data/setup.xml:1: parser error :
  Document is empty
setup.data/setup.xml:1: parser error :
  Start tag expected, '<' not found
Couldn't load 'setup.data/setup.xml'
```

In summary, on 4 out of our 6 test distros, a binary installer for the fifth major release of Google Earth (v5.2.1), a popular commercial application developed by a well-known software company, failed in its *sole goal* of allowing the user to run the application, despite advertising that it should work on any Linux 2.6 machine.

If a team of professional Linux developers had this much trouble getting a widely-used commercial application to be portable across distros, then it is unreasonable to expect researchers or hobbyists to be able to easily create portable Linux packages for their prototypes.

In contrast, once we were able to install Google Earth on just *one machine* (Dell desktop running Ubuntu 8.04), we ran it under CDE supervision to create a self-contained package, copied the package to all 6 test distros, and successfully ran Google Earth on all of them without any installation or configuration.

Benchmark	Native run time	CDE slowdown pack	exec
400.perlbench	23.7s	3.0%	**2.5%**
401.bzip2	47.3s	0.2%	**0.1%**
403.gcc	0.93s	2.7%	**2.2%**
410.bwaves	185.7s	0.2%	**0.3%**
416.gamess	129.9s	0.1%	**0%**
429.mcf	16.2s	2.7%	**0%**
433.milc	15.1s	2%	**0.6%**
434.zeusmp	36.3s	0%	**0%**
435.gromacs	133.9s	0.3%	**0.1%**
436.cactusADM	26.1s	0%	**0%**
437.leslie3d	136.0s	0.1%	**0%**
444.namd	13.9s	3%	**0.3%**
445.gobmk	97.5s	0.4%	**0.2%**
447.dealII	28.7s	0.5%	**0.2%**
450.soplex	5.7s	2.2%	**1.8%**
453.povray	7.8s	2.2%	**1.9%**
454.calculix	1.4s	5%	**4%**
456.hmmer	48.2s	0.2%	**0.1%**
458.sjeng	121.4s	0%	**0.2%**
459.GemsFDTD	55.2s	0.2%	**1.6%**
462.libquantum	1.8s	2%	**0.6%**
464.h264ref	87.2s	0%	**0%**
465.tonto	229.9s	0.8%	**0.4%**
470.lbm	31.9s	0%	**0%**
471.omnetpp	51.0s	0.7%	**0.6%**
473.astar	103.7s	0.2%	**0%**
481.wrf	161.6s	0.2%	**0%**
482.sphinx3	8.8s	3%	**0%**
483.xalancbmk	58.0s	1.2%	**1.8%**

Table 2: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package on the SPEC CPU2006 benchmarks, using the "train" datasets.

7.3 Evaluating CDE run-time slowdown

The primary drawback of executing a CDE-packaged application is the run-time slowdown due to extra user-kernel context switches. Every time the target application issues a system call, the kernel makes two extra context switches to enter and then exit the `cde-exec` monitoring process, respectively. `cde-exec` performs some computations to calculate path redirections, but its run-time overhead is dominated by context switching[6].

We informally evaluated the run-time slowdown of `cde` and `cde-exec` on 34 diverse Linux applications. In summary, for CPU-bound applications, CDE causes almost no slowdown, but for I/O-bound applications, CDE causes a slowdown of up to ~30%.

We first ran CDE on the entire SPEC CPU2006 benchmark suite (both integer and floating-point benchmarks) [8]. We chose this suite because it contains CPU-bound applications that are representative of the types of programs that computational scientists and other researchers are likely to run with CDE. For instance, SPEC CPU2006 contains benchmarks for video compression, molecular dynamics simulation, image ray-tracing, combinatorial optimization, and speech recognition.

We ran these experiments on a Dell machine with a 2.67GHz Intel Xeon CPU running a 64-bit Ubuntu 10.04 distro (Linux 2.6.32). Each trial was run three times, but the variances in running times were negligible.

Table 2 shows the percentage slowdowns incurred by using `cde` to create each package (the 'pack' column) and by using `cde-exec` to execute each package (the 'exec' column). The 'exec' column slowdowns are shown in **bold** since they are more important for our users: A package is only created once but executed multiple times. In sum, slowdowns ranged from non-existent to ~4%, which is unsurprising since the SPEC CPU2006 benchmarks were designed to be CPU-bound and not make much use of system calls.

To test more realistic I/O-bound applications, we measured running times for executing the following commands in the five CDE packages that we created (those labeled with "self" in the "Creator" column of Table 1):

- `pads` — Compile a PADS [19] specification into C code (the "`pads` (compiler)" row in Table 3), and then infer a specification from a data file (the "`pads` (inferencer)" row in Table 3).
- `gadm` — Reproduce the GADM experiment [21]: Compile its C++ source code ('C++ comp'), run genetic algorithm ('algorithm'), and use the R statistics software to visualize output data ('make plots').

Command	Native time	CDE slowdown pack	exec	Syscalls per sec
`gadm` (algorithm)	4187s	0%†	**0%**†	19
`pads` (inferencer)	18.6s	3%†	**1%**†	478
`klee`	7.9s	31%	**2%**†	260
`gadm` (make plots)	7.2s	8%	**2%**†	544
`gadm` (C++ comp)	8.5s	17%	**5%**	1459
`saturn`	222.7s	18%	**18%**	6477
`google-earth`	12.5s	65%	**19%**	7938
`pads` (compiler)	1.7s	59%	**28%**	6969

Table 3: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package. Each entry reports the mean taken over 5 runs; standard deviations are negligible. Slowdowns marked with † are *not* statistically significant at $p < 0.01$ according to a t-test.

[6]Disabling path redirection still results in similar overheads.

- `google-earth` — Measure startup time by launching it and then quitting as soon as the initial Earth image finishes rendering and stabilizes.
- `klee` — Use Klee [16] to symbolically execute a C target program (a STUN server) for 100,000 instructions, which generates 21 test cases.
- `saturn` — Run the regression test suite, which contains 69 tests (each is a static program analysis).

We measured the following on a Dell desktop (2GHz Intel x86, 32-bit) running Ubuntu 8.04 (Linux 2.6.24): number of seconds it took to run the original command ('Native time'), percent slowdown vs. native when running a command with `cde` to create a package ('pack'), and percent slowdown when executing the command from within a CDE package with `cde-exec` ('exec'). We ran each benchmark five times under each condition and report mean running times. We used an *independent two-group t-test* [17] to determine whether each slowdown was statistically significant (i.e., whether the means of two sets of runs differed by a non-trivial amount).

Table 3 shows that the more system calls a program issues per second, the more CDE causes it to slow down due to the extra context switches. Creating a CDE package ('pack' column) is slower than executing a program within a package ('exec' column) because CDE must create new sub-directories and copy files into the package.

CDE execution slowdowns ranged from negligible (not statistically significant) to $\sim$30%, depending on system call frequency. As expected, CPU-bound workloads like the `gadm` genetic algorithm and the `pads` inferencer machine learning algorithm had almost no slowdown, while those that were more I/O- and network-intensive (e.g., `google-earth`) had the largest slowdowns.

When using CDE to run GUI applications, we did not notice any loss in interactivity due to the slowdowns. When we navigated around the 3D maps within the `google-earth` GUI, we felt that the CDE-packaged version was just as responsive as the native version. When we ran GUI programs from CDE packages that users sent to us (the `bio-menace` game, `meld` visual diff tool, and `vr-osg`), we also did not perceive any visible lag.

The main caveat of these experiments is that they are informal and meant to characterize "typical-case" behavior rather than being stress tests of worst-case behavior. One could imagine developing adversarial I/O intensive benchmarks that issue tens or hundreds of thousands of system calls per second, which would lead to greater slowdowns. We have not run such experiments yet.

Finally, we also ran some informal performance tests of `cde-exec`'s seamless execution mode. As expected, there were no noticeable differences in running times versus regular `cde-exec`, since the context-switching overhead dominates `cde-exec` computation overhead.

8 Related work

We know of no published system that automatically creates portable software packages *in situ* from a live running machine like CDE does. Existing tools for creating self-contained applications all require the user to manually specify dependencies at package creation time. For example, Mac OS X programmers can create application bundles using Apple's developer tools IDE [6]. Research prototypes like PDS [14], which creates self-contained Windows apps, and the Collective [23], which aggregates a set of software into a portable *virtual appliance*, also require the user to manually specify dependencies.

VMware ThinApp is a commercial tool that automatically creates self-contained portable Windows applications. However, a user can only create a package by having ThinApp monitor the installation of new software [12]. Unlike CDE, ThinApp cannot be used to create packages from existing software already installed on a live machine, which is our most common use case.

Package management systems are often used to install open-source software and their dependencies. Generic package managers exist for all major operating systems (e.g., RPM for Linux, MacPorts for Mac OS X, Cygwin for Windows), and specialized package managers exist for ecosystems surrounding many programming languages (e.g., CPAN for Perl, RubyGems for Ruby) [4].

From the package creator's perspective, it takes time and expertise to manually bundle up one's software and list all dependencies so that it can be integrated into a specific package management system. A banal but tricky detail that package creators must worry about is adhering to platform-specific idioms for pathnames and avoiding hard-coding non-portable paths into their programs [25]. In contrast, creating a CDE package is as easy as running the target program, and hard-coded paths are fine since `cde-exec` redirects all file accesses into the package.

From the user's perspective, package managers work great as long as the *exact* desired versions of software exist within the system. However, version mismatches and conflicts are common frustrations, and installing new software can lead to a library upgrade that breaks existing software [18]. The Nix package manager is a research project that tries to eliminate dependency conflicts via stricter versioning, but it still requires package creators to manually specify dependencies at creation time [18]. In contrast, CDE packages can be run without any installation, configuration, or risk of breaking existing software.

Virtual machine snapshots achieve CDE's main goal of capturing all dependencies required to execute a set of programs on another machine. However, they require the user to always be working within a VM from the start of a project (or else re-install all of their software within a new VM). Also, VM snapshot disk images are (by defi-

nition) larger than the corresponding CDE packages since they must also contain the OS kernel and other extraneous applications. CDE is a more lightweight solution because it enables users to create and run packages natively on their own machines rather than through a VM.

9 Discussion and conclusions

Our design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to run on other Linux machines with as little effort as possible. However, we are not proposing CDE as a replacement for traditional software installation. CDE packages have a number of limitations. Most notably,

- They are not guaranteed to be complete.
- Their constituent shared libraries are "frozen" and do not receive regular security updates. (Static linking also shares this limitation.)
- They run slower than native applications due to `ptrace` overhead. We measured slowdowns of up to 28% in our informal experiments (§7.3), but slowdowns can be worse for I/O-heavy programs.

Software engineers who are releasing production-quality software should obviously take the time to create and test one-click installers or integrate with package managers. But for the millions of system administrators, research scientists, prototype designers, programming course students and teachers, and hobby hackers who just want to deploy their *ad-hoc* software as quickly as possible, CDE can emulate many of the benefits of traditional software distribution with much less required labor: In just minutes, users can create a base CDE package by running their program under CDE supervision, use our *semi-automated heuristic tools* to make the package complete, deploy to the target Linux machine, and then execute it in *seamless execution mode* to make the target program behave like it was installed normally.

In particular, we believe that the lightweight nature of CDE makes it a useful tool in the Linux system administrator's toolbox. Sysadmins need to rapidly and effectively respond to emergencies, hack together scripts and other utilities on-demand, and run diagnostics without compromising the integrity of production machines. Ad-hoc scripts are notoriously brittle and non-portable across Linux distros due to differences in interpreter versions (e.g., bash vs. dash shell, Python 2.x vs. 3.x), system libraries, and availability of the often-obscure programs that the scripts invoke. Encapsulating scripts and their dependencies within a CDE package can make them portable across distros and minor kernel versions; we have been able to take CDE packages created on 2010-era Linux distros and run them on 2006-era distros [20].

Lessons learned: We would like to conclude by sharing some generalizable system design lessons that we learned throughout the past year of developing CDE.

- First and foremost, start with a conceptually-clear core idea, make it work for basic non-trivial cases, document the still-unimplemented tricky cases, launch your system, and then get feedback from real users. User feedback is by far the easiest way for you to discover what bugs are important to fix and what new features to add next.
- A simple and appealing quick-start webpage guide and screencast video demo are essential for attracting new users. No potential user is going to read through dozens of pages of an academic research paper before deciding to try your system. In short, even hackers need to learn to be great salespeople.
- To maximize your system's usefulness, you must design it to be easy-to-use for beginners but also to allow advanced users to customize it to their liking. One way to accomplish this goal is to have well-designed default settings, which can be adjusted via command-line options or configuration files. The defaults must work well "out-of-the-box" without any tuning, or else beginners will get frustrated.
- Resist the urge to add new features just because they're "interesting", "cool", or "potentially useful". Only add new features when there are compelling real users who demand it. Instead, focus your development efforts on fixing bugs, writing more test cases, improving your documentation, and, most importantly, attracting new users.
- Users are the best sources of bug reports, since they often stress your system in ways that you could have never imagined. Whenever a user reports a bug, try to create a representative minimal test case and add it to your regression test suite.
- If a user has a conceptual misunderstanding of how your system works, then think hard about how you can improve your documentation or default settings to eliminate this misunderstanding.

In sum, get real users, make them happy, and have fun!

Acknowledgments

Special thanks to Dawson Engler for supporting my efforts on this project throughout the past year, to Bill Howe for inspiring me to develop CDE's streaming mode, to Yaroslav Bulatov for being a wonderful CDE power-user and advocate, to Federico D. Sacerdoti (my paper shepherd) for his insightful critiques that greatly improved the prose, and finally to the NSF fellowship for funding this portion of my graduate studies.

References

[1] CDE public source code repository, https://github.com/pgbovine/CDE.

[2] Coq proof assistant: Bug 2443, http://coq.inria.fr/bugs/show_bug.cgi?id=2443.

[3] GCC compiler: Bug 46651, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=46651.

[4] List of software package management systems, http://en.wikipedia.org/wiki/List_of_software_package_management_systems.

[5] LLVM compiler: Bug 8679, http://llvm.org/bugs/show_bug.cgi?id=8679.

[6] Mac OS X Bundle Programming Guide: Introduction, http://developer.apple.com/library/mac/#documentation/CoreFoundation/Conceptual/CFBundles/Introduction/Introduction.html.

[7] Saturn online discussion thread, https://mailman.stanford.edu/pipermail/saturn-discuss/2009-August/000174.html.

[8] Spec cpu2006 benchmarks, http://www.spec.org/cpu2006/.

[9] SSH Filesystem, http://fuse.sourceforge.net/sshfs.html.

[10] arachni project home page, https://github.com/Zapotek/arachni.

[11] graph-tool project home page, http://projects.skewed.de/graph-tool/.

[12] VMware ThinApp User's Guide, http://www.vmware.com/pdf/thinapp46_manual.pdf.

[13] AIKEN, A., BUGRARA, S., DILLIG, I., DILLIG, T., HACKETT, B., AND HAWKINS, P. An overview of the Saturn project. PASTE '07, ACM, pp. 43–48.

[14] ALPERN, B., AUERBACH, J., BALA, V., FRAUENHOFER, T., MUMMERT, T., AND PIGOTT, M. PDS: A virtual execution environment for software deployment. VEE '05, ACM, pp. 175–185.

[15] ALTEKAR, G., AND STOICA, I. ODR: output-deterministic replay for multicore debugging. SOSP '09, ACM, pp. 193–206.

[16] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08, USENIX Association, pp. 209–224.

[17] CHAMBERS, J. M. *Statistical Models in S.* CRC Press, Inc., Boca Raton, FL, USA, 1991.

[18] DOLSTRA, E., DE JONGE, M., AND VISSER, E. Nix: A safe and policy-free system for software deployment. In *LISA '04, the 18th USENIX conference on system administration* (2004).

[19] FISHER, K., AND GRUBER, R. PADS: a domain-specific language for processing ad hoc data. PLDI '05, ACM, pp. 295–304.

[20] GUO, P. J., AND ENGLER, D. CDE: Using system call interposition to automatically create portable software packages (short paper). In *USENIX Annual Technical Conference* (June 2011).

[21] LAHIRI, M., AND CEBRIAN, M. The genetic algorithm as a general diffusion model for social networks. In *Proc. of the 24th AAAI Conference on Artificial Intelligence* (2010), AAAI Press.

[22] LOPER, E., AND BIRD, S. NLTK: The Natural Language Toolkit. In *In ACL Workshop on Effective Tools and Methodologies for Teaching NLP and Computational Linguistics* (2002).

[23] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *LISA '03, the 17th USENIX conference on system administration* (2003).

[24] SCAFFIDI, C., SHAW, M., AND MYERS, B. Estimating the numbers of end users and end user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (2005).

[25] STAELIN, C. mkpkg: A software packaging tool. In *LISA '98, the 12th USENIX conference on system administration* (1998).

[26] SUCAN, I. A., AND KAVRAKI, L. E. Kinodynamic motion planning by interior-exterior cell exploration. In *Int'l Workshop on the Algorithmic Foundations of Robotics* (2008), pp. 449–464.

Improving Virtual Appliance Management through Virtual Layered File Systems

Shaya Potter Jason Nieh
Computer Science Department
Columbia University
{spotter, nieh}@cs.columbia.edu

Abstract

Managing many computers is difficult. Recent virtualization trends exacerbate this problem by making it easy to create and deploy multiple virtual appliances per physical machine, each of which can be configured with different applications and utilities. This results in a huge scaling problem for large organizations as management overhead grows linearly with the number of appliances.

To address this problem, we introduce Strata, a system that combines unioning file system and package management semantics to enable more efficient creation, provisioning and management of virtual appliances. Unlike traditional systems that depend on monolithic file systems, Strata uses a collection of individual sotware layers that are composed together into the Virtual Layered File System (VLFS) to provide the traditional file system view. Individual layers are maintained in a central repository and shared across all file systems that use them. Layer changes and upgrades only need to be done once in the repository and are then automatically propagated to all virtual appliances, resulting in management overhead independent of the number of appliances. Our Strata Linux prototype requires only a single loadable kernel module providing the VLFS support and doesn't require any application or source code level kernel modifications. Using this prototype, we demonstrate how Strata enables fast system provisioning, simplifies system maintenance and upgrades, speeds system recovery from security exploits, and incurs only modest performance overhead.

1 Introduction

A key problem organizations face is how to efficiently provision and maintain the large number of machines deployed throughout their organizations. This problem is exemplified by the growing adoption and use of virtual appliances (VAs). VAs are pre-built software bundles run inside virtual machines (VMs). Since VAs are often tailored to a specific application, these configurations can be smaller and simpler, potentially resulting in reduced resource requirements and more secure deployments.

While VAs simplify application deployment and decrease hardware costs, they can tremendously increase the human cost of administering these machines As VAs are cloned and modified, organizations that once had a few hardware machines to manage now find themselves juggling many more VAs with diverse system configurations and software installations.

This causes many management problems. First, as these VAs share a lot of common data, they are inefficient to store, as there are multiple copies of many common files. Second, by increasing the number of systems in use, we increase the number of systems needing security updates. Finally, machine sprawl, especially non actively maintained machines, can give attackers many places to hide as well as make attack detection more difficult. Instead of a single actively used machine, administrators now have to monitor many irregularly used machines.

Many approaches have been used to address these problems, including diskless clients [5], traditional package management systems [6, 1], copy-on-write disks [9], deduplication [16] and new VM storage formats [12, 4]. Unfortunately, they suffer from various drawbacks that limit their utility and effectiveness in practice. They either do not directly help with management, incur management overheads that grow linearly with the number of VAs, or require a homogenous configuration, eliminating the main advantages of VAs.

The fundamental problem with previous approaches is that they are based on a monolithic file system or block device. These file systems and block devices address their data at the block layer and are simply used as a storage entity. They have no direct concept of what the file system contains or how it is modified. However, managing VAs is essentially done by making changes to the file system. As a result, any upgrade or maintenance operation needs to be done to each VA independently, even when they all need the same maintenance.

We present Strata, a novel system that integrates file system unioning with package management semantics and uses the combination to solve VA management problems. Strata makes VA creation and provisioning fast. It automates the regular maintenance and upgrades that must be performed on provisioned VA instances. Finally,

it improves the ability to detect and recover from security exploits.

Strata achieves this by providing three architectural components: layers, layer repositories, and the Virtual Layered File System (VLFS). A layer is a set of files that are installed and upgraded as a unit. Layers are analogous to software packages in package management systems. Like software packages, a layer may require other layers to function correctly, just as applications often require various system libraries to run. Strata associates dependency information with each layer that defines relationships among distinct layers. Unlike software packages, which are installed into each VA's file system, layers can be shared directly among multiple VAs.

Layer repositories are used to store layers centrally within a virtualization infrastructure, enabling them to be shared among multiple VAs. Layers are updated and maintained in the layer repository. When a new version of an application becomes available, due to added features or a security patch, a new layer is added to the repository. Different versions of the same application may be available through different layers in the layer repository. The layer repository is typically stored in a shared storage infrastructure accessible by the VAs, such as an SAN. Storing layers on the SAN does not impact VA performance because an SAN is where a traditional VA's monolithic file system is stored.

The VLFS implements Strata's unioning mechanism and provides the file system for each VA. Like a traditional unioning file system, it is a collection of individual layers composed into a single view. It enables, a file system to be built out of many shared read-only layers while providing each file system with its own private read-write layer to contain all file system modifications that occur during runtime. In addition, it provides new semantics that enable unioning file systems to be used as the basis for package management type system. These include how layers get added and removed from the union structure as well as how the file system handles files deleted from a read-only layer.

Strata, by combining the unioning and package management semantics, provides a number of management benefits. First, Strata is able to create and provision VAs quickly and easily. By leveraging each layer's dependency information, Strata allows an administrator to quickly create template VAs by only needing to explicitly select the application and tool layers of interest. These template VAs can then be instantly provisioned by end users as no copying or on demand paging is needed to instantiate any file system as all the layers are accessed from the shared layer repository.

Second, Strata automates upgrades and maintenance of provisioned VAs. If a layer contains a bug to be fixed, the administrator only updates the template VA with a replacement layer containing the fix. This automatically informs all provisioned VAs to incorporate the updated layer into their VLFS's namespace view, thereby requiring the fix to only be done once no matter how many VAs are deployed. Unlike traditional VAs, who are updated by replacing an entire file system [12, 4], Strata does not need to be rebooted to have these changes take effect. Unlike package management, all VLFS changes are atomic as no time is spent deleting and copying files.

Finally, this semantic allows Strata to easily recover VAs in the presence of security exploits. The VLFS allows Strata to distinguish between files installed via its package manager, which are stored in a shared read-only layer, and the changes made over time, which are stored in the private read-write layer. If a VA is compromised, the modifications will be confined to the VLFS's private read-write layer, thereby making the changes easy to both identify and remove.

We have implemented a Strata Linux prototype without any application or source code operating system kernel changes and provide the VLFS as a loadable kernel module. We show that by combining traditional package management with file system unioning we provide powerful new functionality that can help automate many machine management tasks. We have used our prototype with VMware ESX virtualization infrastructure to create and manipulate a variety of desktop and server VAs to demonstrate its utility for system provisioning, system maintenance and upgrades, and system recovery. Our experimental results show that Strata can provision VAs in only a few seconds, can upgrade a farm of fifty VAs with several different configurations in less than two minutes, and has scalable storage requirements and modest file system performance overhead.

2 Related Work

The most common way to provision and maintain machines today is using the package management system built into the operating system [6, 1]. Package management provides a number of benefits. First, it divides the installable software into independent chunks called packages. When one wants to install a piece of software or upgrade an already installed piece of software, all one has to do is download and install that single item. Second, these packages can include dependency information that instructs the system about what other packages must be installed with this package. This enables tools [2, 10] to automatically determine the entire set of packages one needs to install when one wants to install a piece of software, making it significantly easier for an end-user to install software.

However, package managers view the file system as a simple container for files and not as a partner in the man-

agement of the machine. This causes them to suffer from a number of flaws in their management of large numbers of VAs. They are not space or time efficient, as each provisioned VA requires time-consuming copying of many megabytes or gigabytes into each VA's file system. These inefficiencies affect both provisioning and updating of a system as a lot of time is spent, downloading, extracting and installing the individual packages into the many independent VAs.

As the package manager does not work in partnership with the file system, the file system does not distinguish between a file installed from a package and a file modified or created in the course of usage. Specialized tools are needed to traverse the entire file system to determine if a file has been modified and therefore compromised. Finally, package management systems work in the context of a running system to modify the file system directly. These tools often cannot not work if the VA is suspended or turned off.

For local scenarios, the size and time efficiencies of provisioning a VA can be improved by utilizing copy-on-write (COW) disks, such as QEMU's QCOW2 [9] format. These enables VAs to be provisioned quickly, as little data has to be written to disk immediately due to the COW property. However, once provisioned, each COW copy is now fully independent from the original, is equivalent to a regular copy, and therefore suffers from all the same maintenance problems as a regular VA. Even if the original disk image is updated, the changes would be incompatible with the cloned COW images. This is because COW disks operate at the block level. As files get modified, they use different blocks on their underlying device. Therefore, it is likely that the original and cloned COW images address the same blocks for different pieces of data. For similar reasons, COW disks do not help with VA creation, as multiple COW disks cannot be combined together into a single disk image.

Both the Collective [4] and Ventana [12] attempt to solve the VA maintenance problem by building upon COW concepts. Both systems enable VAs to be provisioned quickly by performing a COW copy of each VA's `system` file system. However, they suffer from the fact that they manage this file system at either the block device or monolithic file system level, providing users with only a single file system. While ideally an administrator could supply a single homogeneous shared image for all users, in practice, users want access to many heterogeneous images that must be maintained independently and therefore increase the administrator's work. The same is true for VAs provisioned by the end user, while they both enable the VAs to maintain a separate disk from the shared system disk that persists beyond upgrades.

Mirage [17] attempts to improve the disk image sprawl problem by introducing a new storage format, the Mirage Index Format (MIF), to enumerate what files belong to a package. However, it does not help with the actual image sprawl in regard to machine maintenance, because each machine reconstituted by Mirage still has a fully independent file system, as each image has its own personal copy. Although each provisioned machine can be tracked, they are now independent entities and suffer from the same problems as a traditional VA.

Stork [3] improves on package management for container-based systems by enabling containers to hard link to an underlying shared file system so that files are only stored once across all containers. By design, it cannot help with managing independent machines, virtual machines, or VAs, because hard links are a function internal to a specific file system and not usable between separate file systems.

Union file systems [11, 19] provide the ability to compose multiple different file namespaces into a single view. Unioning file systems are commonly used to provide a COW file system from a read-only copy, such as with Live-CDs. However, unioning file system by themselves do not directly help with VA management, as the underlying file system has to be maintained using regular tools. Strata builds upon and leverages this mechanism by improving its ability to handle deleted files as well as managing the layers that belong to the union. This allows Strata to provide a solution that enables efficient provisioning and management of VAs.

Strata focuses on improving virtual appliance management, but the VLFS idea can be used to address other management and security problems as well. For example, our previous work on Apiary [14] demonstrates how the VLFS can be combined with containers to provide a transparent desktop application fault containment architecture that is effective at limiting the damage from exploits to enable quick recovery while being as easy to use as a traditional desktop system.

3 Strata Basics

Figure 1 shows Strata's three architectural components: layers, layer repositories, and VLFSs. A layer is a distinct self-contained set of files that corresponds to a specific functionality. Strata classifies layers into three categories: software layers with self-contained applications and system libraries, configuration layers with configuration file changes for a specific VA, and private layers allowing each provisioned VA to be independent. Layers can be mixed and matched, and may depend on other layers. For example, a single application or system library is not fully independent, but depends on the presence of other layers, such as those that provide needed shared libraries. Strata enables layers to enumerate their dependencies on other layers. This dependency scheme

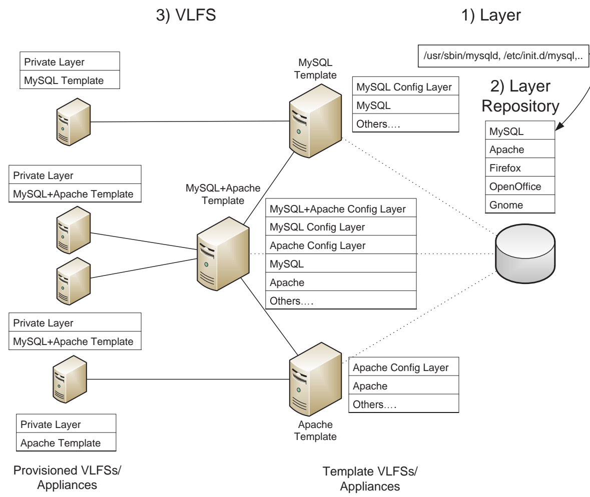


Figure 1: How Strata's Components Fit Together

allows automatic provisioning of a complete, fully consistent file system by selecting the main features desired within the file system.

Layers are provided through layer repositories. As Figure 1 shows, a layer repository is a file system share containing a set of layers made available to VAs. When an update is available, the old layer is not overwritten. Instead, a new version of the layer is created and placed within the repository, making it available to Strata's users. Administrators can also remove layers from the repository, e.g., those with known security holes, to prevent them from being used. Layer repositories are generally stored on centrally managed file systems, such as a SAN or NFS, but they can also be provided by protocols such as FTP and HTTP and mirrored locally. Layers from multiple layer repositories can form a VLFS as long as they are compatible with one another. This allows layers to be provided in a distributed manner. Layers provided by different maintainers can have the same layer names, causing a conflict. This, however, is no different from traditional package management systems as packages with the same package name, but different functionality, can be provided by different package repositories.

As Figure 1 shows, a VLFS is a collection of layers from layer repositories that are composed into a single file system namespace. The layers making up a particular VLFS are defined by the VLFS's layer definition file (LDF), which enumerates all the layers that will be composed into a single VLFS instance. To provision a VLFS, an administrator selects software layers that provide the desired functionality and lists them in the VLFS's LDF.

Within a VLFS, layers are stacked on top of another and composed into a single file system view. An implication of this composition mechanism is that layers on top can obscure files on layers below them, only allowing the contents of the file instance contained within the higher level to be used. This means that files in the private or configuration layers can obscure files in lower layers, such as when one makes a change to a default version of a configuration file located within a software layer. However, to prevent an ambiguous situation from occurring, where the file system's contents depend on the order of the software layers, Strata prevents software layers that contain a subset of the same file from being composed into a single VLFS.

4 Using Strata

Strata's usage model is centered around the usage of layers to quickly create VLFSs for VAs as shown in Figure 1. Strata allows an administrator to compose together layers to form template VAs. These template VAs can be used to form other template appliances that extend their functionality, as well as to provide the VA that end users will provision and use. Strata is designed to be used within the same setup as a traditional VM architecture. This architecture includes a cluster of physical machines that are used to host VM execution as well as a shared SAN that stores all of the VM images. However, instead of storing complete disk images on the SAN, Strata uses the SAN to store the layers that will be used by the VMs it manages.

4.1 Creating Layers and Repositories

Layers are first created and stored in layer repositories. Layer creation is similar to the creation of packages in a traditional package management system, where one builds the software, installs it into a private directory, and turns that directory into a package archive, or in Strata's case, a layer. For instance, to create a layer that contains the MySQL SQL server, the layer maintainer would download the source archive for MySQL, extract it, and build it normally. However, instead of installing it into the system's root directory, one installs it into a virtual root directory that becomes the file system component of this new layer. The layer maintainer then defines the layer's metadata, including its name (`mysql-server` in this case) and an appropriate version number to uniquely identify this layer. Finally, the entire directory structure of the layer is copied into a file system share that provides a layer repository, making the layer available to users of that repository.

4.2 Creating Appliance Templates

Given a layer repository, an administrator can then create template VAs. Creating a template VA involves: (1) Creating the template VA with an identifiable name. (2)

Determining what repositories are available to it. (3) Selecting a set of layers that provide the functionality desired.

To create a template VA that provides a MySQL SQL server, an administrator creates an appliance/VLFS named `sql-server` and selects the layers needed for a fully functional MySQL server file system, most importantly, the mysql-server layer. Strata composes these layers together into the VLFS in a read-only manner along with a read-write private layer, making the VLFS usable within a VM. The administrator boots the VM and makes the appropriate configuration changes to the template VA, storing them within the VLFS's private layer. Finally, the private layer belonging to the template appliance's VLFS is converted into the template's read-only configuration layer by being moved to a SAN file-system that the VAs can only access in a read-only manner. As another example, to create an Apache web server appliance, an administrator creates an appliance/VLFS named `web-server`, and selects the layers required for an Apache web server, most importantly, the layer containing the Apache program.

Strata extends this template model by allowing multiple template VAs to be composed together into a single new template. An administrator can create a new template VA/VLFS, `sql+web-server`, composed of the MySQL and Apache template VAs. The resulting VLFS has the combined set of software layers from both templates, both of their configuration layers, and a new configuration layer containing the configuration state that integrates the two services together, for a total of three configuration layers.

4.3 Provisioning and Running Appliance Instances

In Strata, a VLFS can be created by building off a previously defined VLFS set of layers and combining those layers with a new read-write private layer. Therefore, given previously defined templates, Strata enables VAs to be efficiently and quickly provisioned and deployed by end users. Provisioning a VA involves (1) creating a virtual machine container with a network adapter and an empty virtual disk, (2) using the network adapter's unique MAC address as the machine's identifier for identifying the VLFS created for this machine, and (3) forming the VLFS by referencing the already existing respective template VLFS and combining the template's read-only software and configuration layers with a read-write private layer provided by the VM's virtual disk.

As each VM managed by Strata does not have a physical disk off which to boot, Strata network boots each VM. When the VM boots, its BIOS discovers a network boot server which provides it with a boot image, including a base Strata environment. The VM boots this base environment, which then determines which VLFS should be mounted for the provisioned VM using the MAC address of the machine. Once the proper VLFS is mounted, the machine transitions to using it as its root file system.

4.4 Updating Appliances

Strata upgrades provisioned VAs efficiently using a simple three-step process. First, an updated layer is installed into a shared layer repository. Second, administrators are able to modify the template appliances under their control to incorporate the update. Finally, all provisioned VAs based on that template will automatically incorporate the update as well. Note that updating appliances is much simpler than updating generic machines, as appliances are not independently managed machines. This means that extra software that can conflict with an upgrade will not be installed into a centrally managed appliance. Centrally managed appliance updates are limited to changes to their configuration files and what data files they store.

Strata's updates propagate automatically even if the VA is not currently running. If a provisioned VA is shut down, the VA will compose whatever updates have been applied to its templates automatically, never leaving the file system in a vulnerable state, because it composes its file system afresh each time it boots. If it is suspended, Strata delays the update to when the VA is resumed, as updating layers is a quick task. Updating is significantly quicker than resuming, so this does not add much to its cost.

Furthermore, VAs are upgraded atomically, as Strata adds and removes all the changed layers in a single operation. In contrast, traditional package management system, when upgrading a package, first uninstalls it before reinstalling the newer version. This traditional method leaves the file system in an inconsistent state for a short period of time. For instance, when the libc package is upgraded, its contents are first removed from the file system before being replaced. Any application that tries to execute during the interim will fail to dynamically link because the main library on which it depends is not present within the file system at that moment.

4.5 Improving Security

Strata makes it much easier to manage VAs that have had their security compromised. By dividing a file system into a set of shared read-only layers and storing all file system modifications inside the private read-write layer, Strata separates changes made to the file system via layer management from regular runtime modifications. This enables Strata to easily determine when system files have

been compromised, because making a compromise persistent requires the file system be modified, modifying or adding files to the file system to create a compromise will be readily visible in the private layer. This allows Strata to not rely on tools like Tripwire [8] or maintain separate databases to determine if files have been modified from their installed state. Similarly, this check can be run external to the VA, as it just needs access to the private layer, thereby preventing an attacker from disabling it. This reduces management load due to not requiring any external databases be kept in sync with the file system state as it changes. While an attacker could try to compromise files on the shared layers, they would have to exploit the SAN containing the layer repository. In a regular virtualization architecture, if an attacker could exploit the SAN, he would also have access to all

This segregation of modified file system state also enables quick recovery from a compromised system. By sreplacing the VA's private layer with a fresh private layer, the compromised system is immediately fixed and returned to its default, freshly provisioned state. However, unlike reinstalling a system from scratch, replacing the private layer does not require throwing away the contents of the old private layer. Strata enables the layer to be mounted within the file system, enabling administrators to have easy access to the files located within it to move the uncompromised files back to their proper place.

5 Strata Architecture

Strata introduces the concept of a virtual layered file system in place of traditional monolithic file systems. Strata's VLFS allows file systems to be created by composing layers together into a single file system namespace view. Strata allows these layers to be shared by multiple VLFSs in a read-only manner or to remain read-write and private to a single VLFS.

Every VLFS is defined by a layer definition file, which specifies what software layers should be composed together. An LDF is a simple text file that lists the layers and their respective repositories. The LDF's layer list syntax is `repository/layer version` and can be proceeded by an optional modifier command. When an administrator wants to add or remove software from the file system, instead of modifying the file system directly, they modify the LDF by adding or removing the appropriate layers.

Figure 2 contains an example LDF for a MySQL SQL server template appliance. The LDF lists each individual layer included in the VLFS along with its corresponding repository. Each layer also has a number indicating which version will be composed into the file system. If an updated layer is made available, the LDF is updated

```
main/mysql-server 5.0.51a-3

main/base 1
main/libdb4.2 4.2.52-18
main/apt-utils 0.5.28.6
main/liblocale-gettext-perl 1.01-17
main/libtext-charwidth-perl 0.04-1
main/libtext-iconv-perl 1.2-3
main/libtext-wrapi18n-perl 0.06-1
main/debconf 1.4.30.13
main/tcpd 7.6-8
main/libgdbm3 1.8.3-2
main/perl 5.8.4-8
main/psmisc 21.5-1
main/libssl0.9.7 0.9.7e-3
main/liblockfile1 1.06
main/adduser 3.63
main/libreadline4 4.3-11
main/libnet-daemon-perl 0.38-1
main/libplrpc-perl 0.2017-1
main/libdbi-perl 1.46-6
main/ssmtp 2.61-2
=main/mailx 3a8.1.2-0.20040524cvs-4
```

Figure 2: LDF for MySQL Server Template

to include the new layer version instead of the old one. If the administrator of the VLFS does not want to update the layer, they can hold a layer at a specific version, with the = syntax element. This is demonstrated by the `mailx` layer in Figure 2, which is being held at the version listed in the LDF.

Strata allows an administrator to select explicitly only the few layers corresponding to the exact functionality desired within the file system. Other layers needed in the file system are implicitly selected by the layers' dependencies as described in Section 5.2. Figure 2 shows how Strata distinguishes between explicitly and implicitly selected layers. Explicitly selected layers are listed first and separated from the implicitly selected layers by a blank line. In this case, the MySQL server has only one explicit layer, mysql-server, but has 21 implicitly selected layers. These include utilities such as Perl and TCP Wrappers (tcpd), as well as libraries such as OpenSSL (libssl). Like most operating systems that require a minimal set of packages to always be installed, Strata also always includes a minimal set of shared layers that are common to all VLFSs that it denotes as `base`. In our Strata prototype, these are the layers that correspond to packages that Debian makes essential and are therefore not removable. Strata also distinguishes explicit layers from implicit layers to allow future reconfigurations to remove one implicit layer in favor of another if dependencies need to change.

When an end user provisions an appliance by cloning a template, an LDF is created for the provisioned VA. Fig-

```
@main/sql-server
```

Figure 3: LDF for Provisioned MySQL Server VA

ure 3 shows an example introducing another syntax element, @, that instructs Strata to reference another VLFS's LDF as the basis for this VLFS. This lets Strata clone the referenced VLFS by including its layers within the new VLFS. In this case, because the user wants only to deploy the SQL server template, this VLFS LDF only has to include the single @ line. In general, a VLFS can reference more than one VLFS template, assuming that layer dependencies allow all the layers to coexist.

5.1 Layers

Strata's layers are composed of three components: metadata files, the layer's file system, and configuration scripts. They are stored on disk as a directory tree named by the layer's name and version. For instance, version 5.0.51a of the MySQL server, with a strata layer version of 3, would be stored under the directory `mysql-server_5.0.51a-3`. Within this directory, Strata defines a `metadata` file, a `filesystem` directory, and a `scripts` directory corresponding to the layer's three components.

The metadata files define the information that describes the layer. This includes its name, version, and dependency information. This information is important to ensure that a VLFS is composed correctly. The `metadata` file contains all the metadata that is specified for the layer. Figure 4 shows an example metadata file. Figure 5 shows the full metadata syntax. The metadata file has a single field per line with two elements, the field type and the field contents. In general, the metadata file's syntax is `Field Type: value`, where value can be either a single entry or a comma-separated list of values.

The layer's file system is a self-contained set of files providing a specific functionality. The files are the individual items in the layer that are composed into a larger VLFS. There are no restrictions on the types of files that can be included. They can be regular files, symbolic links, hard links, or device nodes. Similarly, each directory entry can be given whatever permissions are appropriate. A layer can be seen as a directory stored on the shared file system that contains the same file and directory structure that would be created if the individual items were installed into a traditional file system. On a traditional UNIX system, the directory structure would typically contain directories such as `/usr`, `/bin` and `/etc`. Symbolic links work as expected between layers since they work on path names, but one limitation is that hard links cannot exist between layers.

The layer's configuration scripts are run when a layer

```
Layer: mysql-server
Version: 5.0.51a-3
Depends: ..., perl (>= 5.6),
  tcpd (>= 7.6-4),...
```

Figure 4: Metadata for MySQL-Server Layer

```
Layer: Layer Name
Version: Version of Layer Unit
Conflicts: layer1 (opt. constraint), ...
Depends: layer1 (...),
   layer2 (...) | layer3, ...
Pre-Depends: layer1 (...), ...
Provides: virtual_layer, ...
```

Figure 5: Metadata Specification

is added or removed from a VLFS to allow proper integration of the layer within the VLFS. Although many layers are just a collection of files, other layers need to be integrated into the system as a whole. For example, a layer that provides `mp3` file playing capability should register itself with the system's MIME database to allow programs contained within the layer to be launched automatically when a user wants to play an `mp3` file. Similarly, if the layer were removed, it should remove the programs contained within itself from the MIME database.

Strata supports four types of configuration scripts: pre-remove, post-remove, pre-install, and post-install. If they exist in a layer, the appropriate script is run before or after a layer is added or removed. For example, a pre-remove script can be used to shut down a daemon before it is actually removed, while a post-remove script can be used to clean up file system modifications in the private layer. Similarly, a pre-install script can ensure that the file system is as the layer expects, while the post-install script can start daemons included in the layer. The configuration scripts can be written in any scripting language. The layer must include the proper dependencies to ensure that the scripting infrastructure is composed into the file system in order to allow the scripts to run.

5.2 Dependencies

A key Strata metadata element is enumeration of the dependencies that exist between layers. Strata's dependency scheme is heavily influenced by the dependency scheme in Linux distributions such as Debian and Red Hat. In Strata, every layer composed into Strata's VLFS is termed a *layer unit*. Every layer unit is defined by its name and version. Two layer units that have the same name but different layer versions are different units of the same layer. A *layer* refers to the set of layer units of a particular name. Every layer unit in Strata has a set of dependency constraints placed within its metadata. There are four types of dependency constraints: (a) de-

pendency, (b) pre-dependency, (c) conflict and (d) provide.

Dependency and Pre-Dependency: Dependency and pre-dependency constraints are similar in that they require another layer unit to be integrated at the same time as the layer unit that specifies them. They differ only in the order the layer's configuration scripts are executed to integrate them into the VLFS. A regular dependency does not dictate order of integration. A pre-dependency dictates that the dependency has to be integrated before the dependent layer. Figure 4 shows that the MySQL layer depends on TCP Wrappers, (`tcpd`), because it dynamically links against the shared library `libwrap.so.0` provided by TCP Wrappers. MySQL cannot run without this shared library, so the layer units that contain MySQL must depend on a layer unit containing an appropriate version of the shared library. These constraints can also be versioned to further restrict which layer units satisfy the constraint. For example, shared libraries can add functionality that breaks their application binary interface (ABI), breaking in turn any applications that depend on that ABI. Since MySQL is compiled against version 0.7.6 of the libwrap library, the dependency constraint is versioned to ensure that a compatible version of the library is integrated at the same time.

Conflict: Conflict constraints indicate that layer units cannot be integrated into the same VLFS. There are multiple reasons this can occur, but it is generally because they depend on exclusive access to the same operating system resource. This can be a TCP port in the case of an Internet daemon, or two layer units that contain the same file pathnames and therefore would obscure each other. For this reason, Strata defines that two layer units of the same layer are by definition in conflict because they will contain some of the same files.

An example of this constraint occurs when the ABI of a shared library changes without any source code changes, generally due to an ABI change in the tool chain that builds the shared library. Because the ABI has changed, the new version can no longer satisfy any of the previous dependencies. But because nothing else has changed, the file on disk will usually not be renamed either. A new layer must then be created with a different name, ensuring that the library with the new ABI is never used to satisfy an old dependency on the original layer. Because the new layer contains the same files as the old layer, it must conflict with the older layer to ensure that they are not integrated into the same file system.

Provide: Provide dependency constraints introduce virtual layers. A regular layer provides a specific set of files, but a virtual layer indicates that a layer provides a particular piece of general functionality. Layer units that depend on a certain piece of general functionality can depend on a specific virtual layer name in the normal manner, while layer units that provide that functionality will explicitly specify that they do. For example, layer units that provide HTML documentation depend on the presence of a web server to enable a user to view them, but which one is not important. Instead of depending on a particular web server, they depend on the virtual layer name `httpd`. Similarly, layer units containing a web server and obeying system policy for the location of static html content, such as Apache or Boa, are defined to provide the `httpd` virtual layer name and therefore satisfy those dependencies. Unlike regular layer units, virtual layers are not versioned.

Example: Figure 2 shows how dependencies can affect a VLFS in practice. This VLFS has only one explicit layer, mysql-server, but 21 implicitly selected layers. The mysql-server layer itself has a number of direct dependencies, including Perl, TCP Wrappers, and the mailx program. These dependencies in turn depend on the Berkeley DB library and the GNU dbm library, among others. Using its dependency mechanism, Strata is able to automatically resolve all the other layers needed to create a complete file system by specifying just a single layer

Returning to Figure 4, this example defines a subset of the layers that the mysql-server layer requires to be composed into the same VLFS to allow MySQL to run correctly. More generally, Figure 5 shows the complete syntax for the dependency metadata. Provides is the simplest, with only a comma separated list of virtual layer names. Conflicts adds an optional version constraint to each conflicted layer to limit the layer units that are actually in conflict. Depends and Pre-Depends add a boolean OR of multiple layers in their dependency constraints to allow multiple layers to satisfy the dependency.

Resolving Dependencies: To allow an administrator to select only the layers explicitly desired within the VLFS, Strata automatically resolves dependencies to determine which other layers must be included implicitly.

Linux distributions already face this problem and tools have been developed to address it, such as Apt [2] and Smart [10]. To leverage Smart, Strata adopts the same metadata database format that Debian uses for packages for its own layers. In Strata, when an administrator requests that a layer be added to or removed from a template appliance, Smart also evaluates if the operation can succeed and what is the best set of layers to add or remove. Instead of acting directly on the contents of the file system, however, Strata only has to update the template's VLFS's definition file with the set of layers to be composed into the file system.

5.3 Layer Creation

Strata allows layers to be created in two ways. First, Strata allows the `.deb` packages used by Debian-derived distributions and the `.rpm` packages used by RedHat-derived distributions to be converted into layers that Strata users can use. Strata converts packages into layers in two steps. First, Strata extracts the relevant metadata from the package, including its name and version. Second, Strata extracts the package's file contents into a private directory that will be the layer's file system components. When using converted packages, Strata leverages the underlying distribution's tools to run the configuration scripts belonging to the newly created layers correctly. Instead of using the distribution's tools to unpack the software package, Strata composes the layers together and uses the distribution's tools as though the packages have already been unpacked. Although Strata is able to convert packages from different Linux distributions, it cannot mix and match them because they are generally ABI incompatible with one another.

More commonly, Strata leverages existing packaging methodologies to simplify the creation of layers from scratch. In a traditional system, when administrators install a set of files, they copy the files into the correct places in the file system using the root of the file system tree as their starting point. For instance, an administrator might run `make install` to install a piece of software compiled on the local machine. But in Strata layer creation is a three step process. First, instead of copying the files into the root of the local file system, the layer creator installs the files into their own specific directory tree. That is, they make a blank directory to hold a new file system tree that is created by having the `make install` copy the files into a tree rooted at that directory, instead of the actual file system root.

Second, the layer maintainer extracts programs that integrate the files into the underlying file system and creates scripts that run when the layer is added to and removed from the file system. Examples of this include integration with Gnome's GConf configuration system, creation of encryption keys, or creation of new local users and groups for new services that are added. This leverages skills that package maintainers in a traditional package management world already have.

Finally, the layer maintainer needs to set up the metadata correctly. Some elements of the metadata, such as the name of the layer and its version, are simple to set, but dependency information can be much harder. But because package management tools have already had to address this issue, Strata is able to leverage the tools they have built. For example, package management systems have created tools that infer dependencies using an executable dynamically linking against shared libraries [15]. Instead of requiring the layer maintainer to enumerate each shared library dependency, we can programmatically determine which shared libraries are required and populate the dependency fields based on those versions of the library currently installed on the system where the layer is being created.

5.4 Layer Repositories

Strata provides local and remote layer repositories. Local layer repositories are provided by locally accessible file system shares made available by a SAN. They contain layer units to be composed into the VLFS. This is similar to a regular virtualization infrastructure in which all the virtual machines' disks are stored on a shared SAN. Each layer unit is stored as its own directory; a local layer repository contains a set of directories, each of which corresponds to a layer unit. The local layer repository's contents are enumerated in a database file providing a flat representation of the metadata of all the layer units present in the repository. The database file is used for making a list of what layers can be installed and their dependency information. By storing the shared layer repository on the SAN, Strata lets layers be shared securely among different users' appliances. Even if the machine hosting the VLFS is compromised, the read-only layers will stay secure, as the SAN will enforce the read-only semantic independently of the VLFS.

Remote layer repositories are similar to local layer repositories, but are not accessible as file system shares. Instead, they are provided over the Internet, by protocols such as FTP and HTTP, and can be mirrored into a local layer repository. Instead of mirroring the entire remote repository, Strata allows on-demand mirroring, where all the layers provided by the remote repository are accessible to the VAs, but must be mirrored to the local mirror before they can be composed into a VLFS. This allows administrators to store only the needed layers while maintaining access to all the layers and updates that the repository provides. Administrators can also filter which layers should be available to prevent end users from using layers that violate administration policy. In general, an administrator will use these remote layer repositories to provide the majority of layers, much as administrators use a publicly managed package repository from a regular Linux distribution.

Layer repositories let Strata operate within an enterprise environment by handling three distinct yet related issues. First, Strata has to ensure that not all end users have access to every layer available within the enterprise. For instance, administrators may want to restrict certain layers to certain end users for licensing or security reasons. Second, as enterprises get larger, they gain levels of administration. Strata must support the creation of an

enterprise-wide policy while also enabling small groups within the enterprise to provide more localized administration. Third, larger enterprises supporting multiple operating systems cannot rely on a single repository of layers because of inherent incompatibilities among operating systems.

By allowing a VLFS to use multiple repositories, Strata solves these three problems. First, multiple repositories let administrators compartmentalize layers according to the needs of their end users. By providing end users with access only to needed repositories, organizations prevent their end users from using the other layers. Strata depends on traditional file system access control mechanisms to enforce these permissions. Second, by allowing sub-organizations to set up their own repositories, Strata lets a sub-organization's administrator provide the layers that end users need without requiring intervention by administrators of global repositories. Finally, multiple repositories allow Strata to support multiple operating systems, as each distinct operating system has its own set of layer repositories.

5.5 VLFS Composition

To create a VLFS, Strata has to solve a number of file system-related problems. First, Strata has to support the ability to combine numerous distinct file system layers into a single static view. This is equivalent to installing software into a shared read-only file system. Second, because users expect to treat the VLFS as a normal file system, for instance, by creating and modifying files, Strata has to let VLFSs be fully modifiable. By the same token, users must also be able to delete files that exist on the read-only layer.

By basing the VLFS on top of unioning file systems [11, 19], Strata solves all these problems. Unioning file systems join multiple layers into a single namespace. Unioning file systems have been extended to apply attributes such as read-only and read-write to their layers. The VLFS leverages this property to force shared layers to be read-only, while the private layer remains read-write. If a file from a shared read-only layer is modified, it is copied-on-write (COW) to the private read-write layer before it is modified. For example, Live-CDs use this functionality to provide a modifiable file system on top of the read-only file system provided by the CD. Finally, unioning file systems use white-outs to obscure files located on lower layers. For example, if a file located on a read-only layer is deleted, a white-out file will be created on the private read-write layer. This file is interpreted specially by the file-system and is not revealed to the user while also preventing the user from seeing files with the same name.

But end users need to be able to recover deleted files by reinstalling or upgrading the layer containing them. This is equivalent to deleting a file from a traditional monolithic file system, but reinstalling the package containing the file in order to recover it. Also, Strata supports adding and removing layers dynamically without taking the file system off line. This is equivalent to installing, removing, or upgrading a software package while a monolithic file system is online.

Unlike a traditional file system, where deleted system files can be recovered simply by reinstalling the package containing that file, in Strata, white-outs in the private layer persist and continue to obscure the file even if the layer is replaced. To solve this problem, Strata provides a VLFS with additional writeable layers associated with each read-only shared layer. Instead of containing file data, as does the topmost private writeable layer, these layers just contain white-out marks that will obscure files contained within their associated read-only layer. The user can delete a file located in a shared read-only layer, but the deletion only persists for the lifetime of that particular instance of the layer. When a layer is replaced during an upgrade or reinstall, a new empty white-out layer will be associated with the replacement, thereby removing any preexisting white-outs. In a similar way, Strata handles he case where a file belonging to a shared read-only layer is modified and therefore copied to the VLFS's private read-write layer. Strata provides a *revert* command that lets the owner of a file that has been modified revert the file to its original pristine state. While a regular VLFS *unlink* operation would have removed the modified file from the private layer and created a white-out mark to obscure the original file, *revert* only removes the copy in the private layer, thereby revealing the original below it.

Strata also allows a VLFS to be managed while being used. Some upgrades, specifically of the kernel, will require the VA to be rebooted, but most should be able to occur without taking the VA off line. However, if a layer is removed from a union, the data is effectively removed as well because unions operate only on file system namespaces and not on the data the underlying files contain. If an administrator wants to remove a layer from the VLFS, they must take the VA off line, because layers cannot be removed while in use.

To solve this problem, Strata emulates a traditional monolithic file system. When an administrator deletes a package containing files in use, the processes that are currently using those files will continue to work. This occurs by virtue of `unlink`'s semantic of first removing a file from the file system's namespace, and only removing its data after the file is no longer in use. This lets processes continue to run because the files they need will not be removed until after the process terminates. This creates a semantic in which a currently running program

can be using versions of files no longer available to other programs.

Existing package managers use this semantic to allow a system to be upgraded online, and it is widely understood. Strata applies the same semantic to layers. When a layer is removed from a VLFS, Strata marks the layer as `unlinked`, removing it from the file system namespace. Although this layer is no longer part of the file system namespace and thus cannot be used by any operations such as `open` that work on the namespace, it does remain part of the VLFS, enabling data operations such as `read` and `write` to continue working correctly for previously opened files.

6 Experimental Results

We have implemented Strata's VLFS as a loadable kernel module on an unmodified Linux 2.6 series kernel as well as a set of userspace management tools. The file system is a stackable file system and is an extended version of UnionFS [19]. We present experimental results using our Strata Linux prototype to manage various VAs, demonstrating its ability to reduce management costs while incurring only modest performance overhead. Experiments were conducted on VMware ESX 3.0 running on an IBM BladeCenter with 14 IBM HS20 eServer blades with dual 3.06 GHz Intel Xeon CPUs, 2.5 GB RAM, and a Q-Logic Fibre Channel 2312 host bus adapter connected to an IBM ESS Shark SAN with 1 TB of disk space. The blades were connected by a gigabit Ethernet switch. This is a typical virtualization infrastructure in an enterprise computing environment where all virtual machines are centrally stored and run. We compare plain Linux VMs with a virtual block device stored on the SAN and formatted with the ext3 file system to VMs managed by Strata with the layer repository also stored on the SAN. By storing both the plain VM's virtual block device and Strata's layers on the SAN, we eliminate any differences in performance due to hardware architecture.

To measure management costs, we quantify the time taken by two common tasks, provisioning and updating VAs. We quantify the storage and time costs for provisioning many VAs and the performance overhead for running various benchmarks using the VAs. We ran experiments on five VAs: an Apache web server, a MySQL SQL server, a Samba file server, an SSH server providing remote access, and a remote desktop server providing a complete GNOME desktop environment. While the server VAs had relatively few layers, the desktop VA has very many layers. This enables the experiments to show how the VLFS performance scales as the number of layers increases. To provide a basis for comparison, we provisioned these VAs using (1) the normal VMware virtualization infrastructure and plain Debian package management tools, and (2) Strata. To make a conservative comparison to plain VAs and to test larger numbers of plain VAs in parallel, we minimized the disk usage of the VAs. The desktop VA used a 2 GB virtual disk, while all others used a 1 GB virtual disk.

	Apache	**MySQL**	**Samba**	**SSH**	**Desktop**
Plain	184s	179s	183s	174s	355s
Strata	0.002s	0.002s	0.002s	0.002s	0.002s
QCOW2	0.003s	0.003s	0.003s	0.003s	0.003s

Table 1: VA Provisioning Times

6.1 Reducing Provisioning Times

Table 1 shows how long it takes Strata to provision VAs versus regular and COW copying. To provision a VA using Strata, Strata copies a default VMware VM with an empty sparse virtual disk and provides it with a unique MAC address. It then creates a symbolic link on the shared file system from a file named by the MAC address to the layer definition file that defines the configuration of the VA. When the VA boots, it accesses the file denoted by its MAC address, mounts the VLFS with the appropriate layers, and continues execution from within it. To provision a plain VA using regular methods, we use QEMU's `qemu-img` tool to create both raw copies and COW copies in the QCOW2 disk image format.

Our measurements for all five VAs show that using COW copies and Strata takes about the same amount of time to provision VAs, while creating a raw image takes much longer. Creating a raw image for a VAs takes 3 to almost 6 minutes and is dominated by the cost of copying data to create a new instance of the VA. For larger VAs, these provisioning times would only get worse. In contrast, Strata provisions VAs in only a few milliseconds because a null VMware VM has essentially no data to copy. Layers do not need to be copied, so copying overhead is essentially zero. While COW images can be created in a similar amount of time, they do not provide any of the management benefits of Strata, as each new COW image is independent of the base image from which it was created.

6.2 Reducing Update Times

Table 2 shows how long it takes to update VAs using Strata versus traditional package management. We provisioned ten VA instances each of Apache, MySQL, Samba, SSH, and Desktop for a total of 50 provisioned VAs. All were kept in a suspended state. When a security patch was made available for the `tar` package installed in all the VAs, we updated them [18]. Strata simply updates the layer definition files of the VM templates, which it can do even when the VAs are not active. When the VA is later resumed during normal operation,

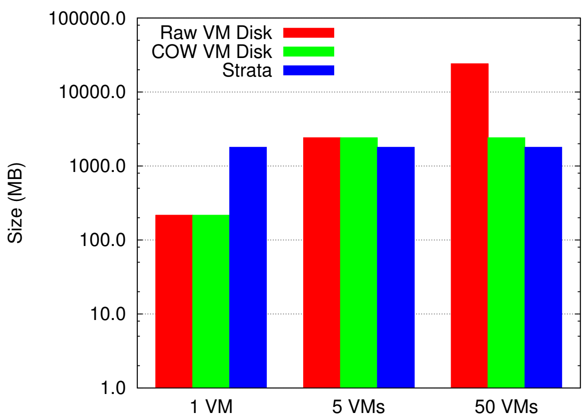


Figure 6: Storage Requirements

it automatically checks to see if the layer definition file has been updated and updates the VLFS namespace view accordingly, an operation that is measured in microseconds. To update a plain VA using normal package management tools, each VA instance needs to be resumed and put on the network. An administrator or script must ssh into each VA, fetch and install the update packages from a local Debian mirror, and finally re-suspend the VA.

Table 2 shows the total average time to update each VA using traditional methods versus Strata. We break down the update time into times to resume the VM, get access to the network, actually perform the update, and re-suspend the VA. The measurements show that the cost of performing an update is dominated by the management overhead of preparing the VAs to be updated and not the update itself. Preparation is itself dominated by getting an IP address and becoming accessible on a busy network. While this cost is not excessive on a quiet network, on a busy network it can take a significant amount of time for the client to get a DHCP address, and for the ARP on the machine controlling the update to find the target machine. The average total time to update each plain VA is about 73 seconds. In contrast, Strata takes only a second to update each VA. As this is an order of magnitude shorter even than resuming the VA, Strata is able to delay the update to a point when the VA will be resumed from standby normally without impacting its ability to quickly respond. Strata provides over 70 times faster update times than traditional package management when managing even a modest number of VAs. Strata's ability to decrease update times would only improve as the number of VAs being managed grows.

	Plain	Strata
VM Wake	14.66s	NA
Network	43.72s	NA
Update	10.22s	1.041s
Suspend	3.96s	NA
Total	73.2s	1.041s

Table 2: VA Update Times

6.3 Reducing Storage Costs

Figure 6 shows the total storage space required for different numbers of VAs stored with raw and COW disk images versus Strata. We show the total storage space for 1 Apache VA, 5 VAs corresponding to an Apache, MySQL, Samba, SSH, and Desktop VA, and 50 VAs corresponding to 10 instances of each of the 5 VAs. As expected, for raw images, the total storage space required grows linearly with the number of VA instances. In contrast, the total storage space using COW disk images and Strata is relatively constant and independent of the number of VA instances. For one VA, the storage space required for the disk image is less than the storage space required for Strata, as the layer repository used contains more layers than those used by any one of the VAs. In fact, to run a single VA, the layer repository size could be trimmed down to the same size as the traditional VA.

For larger numbers of VAs, however, Strata provides a substantial reduction in the storage space required, because many VAs share layers and do not require duplicate storage. For 50 VAs, Strata reduces the storage space required by an order of magnitude over the raw disk images. Table 3 shows that there is much duplication among statically provisioned virtual machines, as the layer repository of 405 distinct layers needed to build the different VLFSs for multiple services is basically the same size as the largest service. Although initially Strata does not have an significant storage benefit over COW disk images, as each COW disk image is independent from the version it was created from, it now must be managed independently. This increases storage usage, as the same updates must be independently applied to many independent disk images

6.4 Virtualization Overhead

To measure the virtualization cost of Strata's VLFS, we used a range of micro-benchmarks and real application workloads to measure the performance of our Linux Strata prototype, then compared the results against vanilla Linux systems within a virtual machine. The virtual machine's local file system was formatted with the `Ext3` file system and given read-only access to a SAN partition formatted with `Ext3` as well. We performed each benchmark in each scenario 5 times and provide the average of the results.

Repo	Apache	MySQL	Samba	SSH	Desktop
1.8GB	217MB	206MB	169MB	127MB	1.7GB
# Layer	43	23	30	12	404
Shared	191MB	162MB	152MB	123MB	169MB
Unique	26MB	44MB	17MB	4MB	1.6GB

Table 3: Layer Repository vs. Static VAs

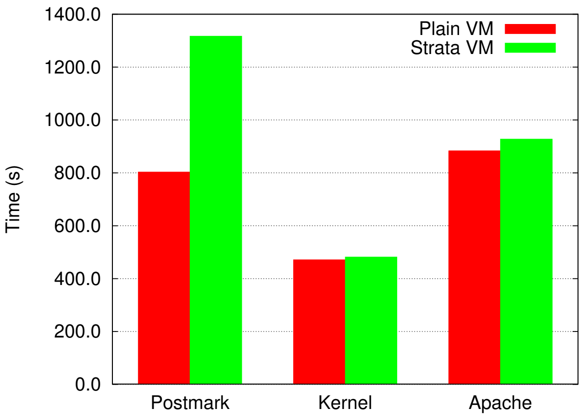


Figure 7: Application Benchmarks

To demonstrate the effect that Strata's VLFS has on system performance, we performed a number of benchmarks. Postmark [7], the first benchmark, is a synthetic test that measures how the system would behave if used as a mail server. Our postmark test operated on files between 512 and 10K bytes, with an initial set of 20,000 files, and performed 200,000 transactions. Postmark is very intensive on a few specific file system operations such as `lookup()`, `create()`, and `unlink()`, because it is constantly creating, opening, and removing files. Figure 7 shows that running this benchmark within a traditional VA is significantly faster than running it in Strata. This is because as Strata composes multiple file system namespaces together, it places significant overhead on those namespace operations.

To demonstrate that postmark's results are not indicative of application oriented performance, we ran two application benchmarks to measure the overhead Strata imposes in a desktop and server VA scenario. The first benchmark was a multi-threaded build of the Linux 2.6.18.6 kernel with two concurrent jobs using the two CPUs allocated to the VM. In all scenarios, we added the 8 software layers required to build a kernel to the layers needed to provide the service. Figure 7 shows that while Strata imposes a slight overhead on the kernel build compared to the underlying file system it uses, the cost is minimal, under 5% at worst.

The second benchmark measured the amount of HTTP transactions that were able to be completed per second to an Apache web server placed under load. We imported the database of a popular guitar tab search engine and used the `http_load` [13] benchmark to continuously performed a set of 20 search queries on the database until 60,000 queries in total have been performed. For each case that did not already contain Apache, we added the appropriate layers to the layer definition file to make Apache available. Figure 7 shows that Strata imposes a minimal overhead of only 5%.

While the Postmark benchmark demonstrated that the VLFS is not an appropriate file system for workloads that are heavy with namespace operations, this shouldn't prevent Strata from being used in those scenarios. No file system is appropriate for all workloads and no system has to be restricted to simply using one file system. One can use Strata and the VLFS to manage the system's configuration while also providing an additional traditional file system on a seperate partition or virtual disk drive to avoid all the overhead the VLFS imposes. This will be very effective for workloads, such as the mail server Postmark is emulating, where namespace heavy operations, such as a mail server processing its mail queue, can be kept on a dedicated file system.

7 Conclusions and Future Work

Strata introduces a new and better way for system administrators to manage virtual appliances using virtual layered file systems. Strata integrates package management semantics with the file system by using a novel form of file system unioning enable dynamic composition of file system layers. This provides powerful new management functionality for provisioning, upgrading, securing, and composing VAs. VAs can be quickly and simply provisioned as no data needs to be copied into place. VAs can be easily upgraded as upgrades can be done once centrally and applied atomically, even for a heterogeneous mix of VAs and when VAs are suspended or turned off. VAs can be more effectively secured since file system modifications are isolated so compromises can be easily identified. VAs can be composed as building blocks to create new systems since file system composition also serves as the core mechanism for creating and maintaining VAs. We have implemented Strata on Linux by providing the VLFS as a loadable kernel modules, but without requiring any source code level kernel changes, and have demonstrated how a Strata can be used in real life situations to improve the ability of system administrators to manage systems. Strata significantly reduces the amount of disk space required for multiple VAs, and allows them to be provisioned almost instantaneously and quickly updated no matter how many are in use.

While Strata just exists as a lab prototype today, there are few steps that could make it significantly more deployable. First, our changes to UnionFS should either be integrated with the current version of UnionFS or with another unioning file system. Second, better tools should be created for managing the creation and management of individual layers. This can include better tools for converting layers from existing Linux distributions as well as new tools that enable layers to be created in a way that takes full advantage of Strata's concepts. Third, the ability to to integrate Strata's concepts with cloud com-

puting infrastructures, such as Eucalyptus, should be investigated.

Acknowledgments

Carolyn Rowland provided helpful comments on earlier drafts of this paper. This work was supported in part by AFOSR MURI grant FA9550-07-1-0527 and NSF grants CNS-1018355, CNS-0914845, and CNS-0905246.

References

[1] The RPM Package Manager. http://www.rpm.org/.

[2] B. Byfield. An Apt-Get Primer. http://www.linux.com/articles/40745, Dec. 2004.

[3] J. Capps, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman. Stork: Package Management for Distributed VM Environments. In *The 21st Large Installation System Administration Conference*, Dallas, TX, Nov. 2007.

[4] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *The 2nd Symposium on Networked Systems Design and Implementation*, pages 259–272, Boston, MA, Apr. 2005.

[5] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[6] J. Fernandez-Sanguino. Debian GNU/Linux FAQ - Chapter 8 - The Debian Package Management Tools. http://www.debian.org/doc/FAQ/ch-pkgtools.en.html.

[7] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR3022, Network Appliance, Inc., Oct. 1997.

[8] G. Kim and E. Spafford. Experience with Tripwire: Using Integrity Checkers for Intrusion Detection. In *The 1994 System Administration, Networking, and Security Conference*, Washington, DC, Apr. 1994.

[9] M. McLoughlin. QCOW2 Image Format. http://www.gnome.org/~markmc/qcow-image-format.htm, Sept. 2008.

[10] G. Niemeyer. Smart Package Manager. http://labix.org/smart.

[11] J.-S. Pendry and M. K. McKusick. Union Mounts in 4.4BSD-lite. In *The 1995 USENIX Technical Conference*, New Orleans, LA, Jan. 1995.

[12] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *3rd Symposium on Networked Systems Design and Implementation*, pages 353–366, San Jose, CA, May 2006.

[13] J. Poskanzer. http://www.acme.com/software/http_load/.

[14] S. Potter and J. Nieh. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *The 2010 USENIX Annual Technical Conference*, pages 103–116, June 2010.

[15] D. Project. DDP Developers' Manuals. http://www.debian.org/doc/devel-manuals.

[16] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *1st USENIX conference on File and Storage Technologies*, Monterey, CA, Jan. 2002.

[17] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl. In *The 2008 ACM International Conference on Virtual Execution Environments*, pages 111–120, Seattle, WA, Mar. 2008.

[18] F. Weimer. DSA-1438-1 Tar – Several Vulnerabilities. http://www.ua.debian.org/security/2007/dsa-1438, Dec. 2007.

[19] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage*, 2(1):1–32, Feb. 2006.

Sequencer: smart control of hardware and software components in clusters (and beyond).

Dr. Pierre Vignéras
pierre.vigneras@bull.net
Extreme Computing R&D
Bull, Architect of an Open World
www.bull.com

September 15, 2011

Abstract

Starting/stopping a whole cluster or a part of it is a real challenge considering the different commands related to various device types and manufacturers, and the order that should be respected. This article presents a solution called the *sequencer* that allows the automatic shutting down and starting up of clusters, subset of clusters or even data-centers. It provides two operation modes designed for ease of use and emergency conditions. Our product has been designed to be efficient and it is currently used to power on and power off one of the largest cluster in the world: the Tera-100, made of more than 4000 nodes.

Keywords: emergency power off, start/stop procedure, actions sequencing, workflow, planning, cluster management, automation, case study.

1 Introduction

Emergency Power Off (EPO) is often not considered from a system administration point of view in traditionnal clusters[1]. Even for Tier-IV infrastructures [1], EPO may happen for various reasons. In such cases, stopping (or starting, both cases are addressed) macro components such as a whole rack or a rack set requires an appropriate sequence of actions. Considering the vast number of different components a cluster is composed of:

nodes: compute nodes[2], login nodes, management nodes, io (nfs, lustre, ...) nodes, ...

hardware: power switches (also called Power Distribution Units or PDUs), ethernet switches, infiniband [2] switches, cold doors[3], disk arrays, ...

powering on/off a whole set of racks can be a real challenge.

First, since it is made of a set of heterogeneous devices, starting/stopping each component of a cluster is not straightforward: usually each device type comes with its own poweron/off command. For example, shutting down a node can be as simple as an

[1] In this article, we consider clusters because they are the first target of our solution. However, this solution also applies to general data-centers.

[2] From a hardware perspective, a node in a cluster is just a computer. A distinction is made however between nodes depending on their roles in the cluster. For example, user might connect to a login node for development, and job submission. The batch scheduler runs on the management node and dispatch jobs to compute nodes. Compute nodes access storage through io nodes and so on.

[3] A cold door is a water-based cooling system produced by Bull that allows high density server in the order of 40 kW per rack.

'ssh host /sbin/halt -p'. However, it might be preferable to use an out of band command through the IPMI BMC[4] if the node is unresponsive for example. Starting a node can be done using a wake on lan [3] command or an IPMI [4] command. Some devices cannot be powered on/off remotely (infiniband or ethernet switches for example). Those devices might be connected to manageable Power Distribution Units (PDUs) that can remotely switch their outlets on/off using SNMP [5] commands. On the extreme case, manual intervention might be required to switch on/off the electrical power.

For software components, there is also a need to manage the shutdown of multiple components on different nodes. Considering high availability framework, virtualization, localization, and clients, using standard calls to `/etc/init.d/service [start|stop]` is often inappropriate.

Finally, the set of instructions for the powering on/off of each cluster's components should be ordered. Trivial examples include:

- powering off an ethernet switch too soon may prevent other components, including nodes, from being powered off;
- powering off a cold door should be done at the very end to prevent cooled components from being burnt out.

By the way, this ordering problem is not only relevant to hardware devices. A software component can also require that a sequence of instructions is executed before being stopped. As a trivial example, when an NFS daemon is stopped, one may prefer that all NFS clients unmount their related directories first in order to prevent either the fill of syslog with NFS mount error (when NFS mount option is 'soft') or the load average brutal increase due to the freezing of softwares accessing the NFS directories (when NFS mount option is 'hard').

Therefore, in this article, the generic term 'component' may define a hardware component such as a node, a switch, or a cold door, or a software component such as a lustre server or an NFS server.

[4]Baseboard Management Controller.

Our proposition — called the *sequencer* — addresses the problem of starting/stopping a cluster (or a data-center). Its design takes into account emergency conditions. Those conditions impose various constraints addressed by our solution:

- Predictive: an EPO should have been validated before being used. It should not perform unknown actions.
- Easy: an EPO should be easy to launch. The emergency cause may happen at any time, especially when skilled staff is not present. Therefore, the EPO procedure should be launchable by "unskilled" humans.
- Fast: an EPO should be as fast as possible.
- Smart: an EPO should power off each component of a cluster in the correct order so most resources will be preserved.
- Robust: an EPO should be tolerant to failure. For example, if a shutdown on a node cooled by a cold door returned an error, the corresponding cold door should not be switched off to prevent the burnout of the node. On the other side, the rest of the cluster can continue the EPO process.

This article is organized as follow: section 2 exposes the design of our solution while some implementation details are dealt with in section 3 following by scalability issues in section 4. Some results of our initial implementation are given section 5. Section 6 compares our solution to related works. Finally, section 7 presents future works.

2 Design

Three major difficulties arise when considering the starting/stopping of a cluster or of a subset of it:

1. the computing of the dependency graph between components (power off a cold door after all components of the related rack have been powered off);

2. the defining of an efficient (scalable) instructions sequence where the order defined by the dependency graph is respected (powering off nodes might be done in parallel);

3. the execution of the instructions set itself, taking failure into account properly (do not power off the cold door, if related rack's nodes have failed to power off).

Therefore, the sequencer is made of three distinct functional layers:

Dependency Graph Maker (DGM): this layer computes the dependency graph according to dependency rules defined by the system administrator in a database.

Instructions Sequence Maker (ISM): this layer computes the instructions sequence that should be executed for starting/stopping the given list of components and that satisfies dependency constraints defined in the dependency graph computed by the previous layer.

Instructions Sequence Executor (ISE): this layer executes the instructions sequence computed by the previous layer and manages the handling of failures.

Finally, a chaining of those layers is supported through an "all-in-one" command.

This design provides therefore two distinct processes for the starting/stopping of components:

Incremental Mode: in this mode, each stage is run separately. The output of each stage can be verified and modified before being passed to the next stage as shown on figure 2.1. The incremental mode generates a script from constraints expressed in a database table and from a components list. This script is optimized in terms of scalability and is intepretable by the instructions sequence executor that deals with parallelism and failures. This mode is the one designed for emergency cases. The instructions set computed should be validated before being used in production.

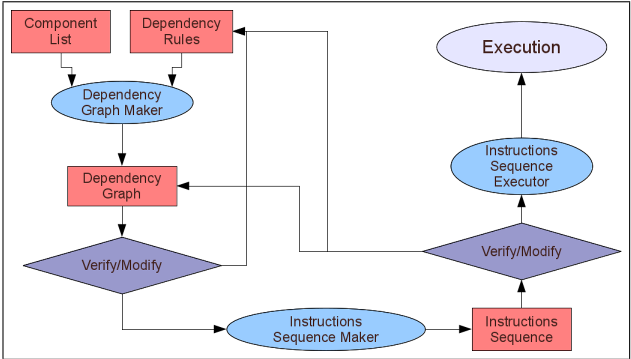


Figure 2.1: Incremental Mode: each stage output can be verified and modified before being passed to the next one.

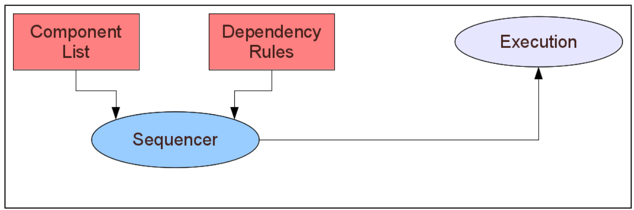


Figure 2.2: Black Box Mode: using the sequencer for simple non-critical usage.

Black Box Mode: in this mode, illustrated in figure 2.2, chaining feature is used to start/stop components as shown by the following syntax:

```
# clmsequencer \              # command name
stop \                        # ruleset name
colddoor3 node[100-200]       # components list
```

This command is somewhat equivalent to the following:

```
# clmsequencer \              # command name
depmake \                     # dgm stage
stop \                        # ruleset name
colddoor3 node[100-200] \     # components list
|clmsequencer seqmake \       # ism stage
|clmsequencer seqexec         # ise stage
```

and can therefore be seen as a syntactic sugar.

The computation of the dependency graph and of the instruction set can take a significant amount of time, especially on very large clusters such as the

Tera-100[5]. This is another good reason for choosing the incremental mode in emergency conditions where each minute is important.

3 Implementation

3.1 Dependency Graph Maker (DGM)

The Dependency Graph Maker (DGM) is the first stage of the sequencer. It takes a components list in parameter, and produces a dependency graph in output. It uses a set of dependency rules described in a database table. The CLI has the following usage:

```
# clmsequencer depgraph [--out file]
ruleset cl_1...cl_N
```

The output is a human readable description of the computed dependency graph in XML format that the Instructions Sequence Maker can parse. By default, the computed dependency graph is produced on the standard output.

The `--output file` option allows the computed dependency graph to get written in the specified `file`.

The `ruleset` parameter defines which ruleset should be used to compute the dependency graph. Ruleset will be explained in section 3.1.2 on the sequencer table.

Finally, other parameters `cl_1...cl_N` define on which components the dependency graph should be computed. Each parameter describes a list of component in a specific format describes in next section 3.1.1.

3.1.1 Components list specification

The first stage of the sequencer takes as an input a list of components. This list is of the form:

```
prefix[a-b,c-d,...][#type][@category]
```

where:

prefix[a-b,c-d,...]: is the standard contracted notation for designing a set of names prefixed by `'prefix'` and suffixed by a number taken in the range given by intervals $[a-b]$, $[c-d]$, and so on. For example, `compute[1-3,5,7-8]` defines names: `compute1`, `compute2`, `compute3`, `compute5`, `compute7`, `compute8`.

category: is optionnal and defines the table[6] where given names should be looked for their `type` (if not given). The type of a component is used in the definition of the dependency table as described in section 3.1.2. Category examples (with some related types) are: `node` (`io`, `nfs`, `login,compute`), `hwmanager` (`bmc`, `cmc`, `coldoor`[7]) and `soft` (`nfsd`, `nagios`, `sshd`).

Some examples of full component list names are given below:

`R-[1-3]#io@rack`: the io racks R-1, R-2 and R-3;

`bullx[10-11]#mds@node`: the lustre mds node bullx10 and bullx11;

`colddoor1#coldoor@hwmanager`: the cold door numbered 1;

`esw-1#eth@switch`: the ethernet switch esw-1;

`server[1-2]#nfsd@soft`: NFS daemons running on server1 and server2.

3.1.2 Sequencer Dependency Rules: the sequencer table

The Dependency Graph Maker requires dependency rules to be specified in a database table. This table describes multiple sets of dependency rules. A *ruleset* is defined as a set of dependency rules. For example, there is one ruleset called `smartstart` containing all

[5] Tera-100 is ranked #6 in the Top500 november 2010 list of fastest supercomputers in the world and #1 in Europe. It is composed of several thousands of Bull bullx series S servers. See `http://www.top500.org/` for details.

[6] It is considered a good practice to have a database where the cluster is described. In a bullx cluster, each component is known and various informations are linked to it such as its model, its status, its location and so on. There should be a way to find a type from a component name. In this article, we use a database for that purpose, any other means can be used though.

[7] Cold doors are spelled 'coldoor' in bullx cluster database.

the dependency rules required for the starting of components. Another ruleset containing all dependency rules required for the stopping of components would be called `smartstop`.

The format of this table is presented below. One line in the table represents exactly one dependency rule. Table columns are:

`ruleset`: the name of the ruleset this dependency rule is a member of.

`name`: the rule name, this name is used as a reference in the `dependson` column, it should be unique in the ruleset;

`types`: the component types the rule should be applied to. A type is specified using the full name (that is, `'type@category'`). Multiple types should be separated by the "pipe" symbol as in `compute@node|io@node`. The special string `'ALL'` acts like a joker: `'ALL@node'` means any component from table node matches, while `'ALL@ALL'` means any component matches, and is equivalent to `'ALL'` alone.

`filter`: an expression of the following two forms:

- `%var =~ regexp`
- `%var !~ regexp`

where `'%var'` is a variable that will be replaced by its value on execution (see table 1 for the list of available variables). The operator `'=~'` means that component will be filtered in only if a match occurs while `'!~'` means the component will be filtered in only if a match does not occur (said otherwise, it a match occurs, it will be filtered out).

If the expression does not start with a known `'%var'` then, the expression is interpreted as a (shell) command that when called specifies if the given component should be filtered in (returned code is 0) or out (returned code is different than 0). Variables will also be replaced before command execution, if specified. As an example, to filter out any component which name starts with the string `'bullx104'`, one would use: `'%name =~ ^bullx104'`. On the other side, to let a script decide on the component id, one would use: `'/usr/bin/my_filter %id'`.

Finally, two special values are reserved for special meanings here:

- String `'ALL'`: any component is filtered in (i.e. accepted);
- The `'NULL'` special DB value: any component is filtered out (i.e. refused).

`action`: the (shell) command that should be executed for each component that matches the rule type (and that have been filtered in). Variables will be replaced, if specified (see table 1 for the list of available variables). If the action is prefixed with the `'@'` symbol, the given action will be executed on the component using an `'ssh'` internal connexion. Depending on the action exit code, the Instruction Sequence Executor may continue its execution, or abort. This will be discussed in section 3.3.

`depsfinder`: the (shell) command that specifies which components the current component depends on. The command should return the components set on its standard output, one component per line. A component should be of the following format: `'name#type@category'`. Variables will be replaced, if specified (see table 1 for the list of available variables). When set to the `'NULL'` special DB value, rule names specified in the `dependson` column are simply ignored.

`dependson`: a comma-separated list of rule names, this rule depends on. For each dependency returned by the depsfinder, the sequencer looks if the dependency type matches one of the rule type specified by this field (rule names specified should be in the same ruleset). If such a match occurs, the rule is applied on the dependency. When set to the `'NULL'` special DB value, the script specified in the `'depsfinder'` column is simply ignored.

`comments`: a free form comment.

Name	Value	Example
%id	The full name of the component	bullx12#compute@node
%name	The name of the component	bullx12
%type	The type of the component	compute
%category	The category of the component	node
%ruleset	The current ruleset being processed	smartstop
%rulename	The current rule being processed	compute_off

Table 1: List of available variables.

The framework does not allow the specification of a timeout for a given action for two main reasons:

1. Granularity: if such a specification was provided at that level (in the sequencer table), the timeout would be specified for all components type specified by the `'type'` column whereas it seems preferable to have a lower granularity, per component. This is easily achievable by the action script itself for which the component can be given as a parameter.
2. The action script, for a given component, knows what to do when a timeout occurs much better that the sequencer itself. Therefore, if a specific process is required after a command timeout (such as a retry), the action script should implement itself the required behavior when the timeout occurs and returns the appropriate return code.

As an example we will consider the sequencer table presented in table 2.

3.1.3 Algorithm

The objective of the Dependency Graph Maker is to output the dependency graph based on the dependency rules defined in the related table and on the components list given as a parameter. The computing of the dependency graph involves the following steps:

1. **Components List Expansion**: from the given components list, the expansion should be done. It returns a list of names of the form: `'name#type@category'`. Such name is called *id* in the following.
2. **Dependency Graph Creation**: the dependency graph is created as a set of disconnected nodes where each node is taken from the list of ids. A node in the dependency graph has the following form: `'id [actionsList]'` where `'actionsList'` is the list of actions that should be executed for the component with the corresponding `'id'`. This graph is updated during the process by:

   (a) Node additions: when processing a given component `'c'`, through a dependency rule (one row in the related ruleset table), the command line specified by column `'depsfinder'` is executed. This execution may return a list of components that should be processed before the current one. Each of those components will therefore be added to the dependency graph if it is not already present.

   (b) Arc additions: for each components returned by the `'depsfinder'` script, an arc is added between `'c'` and that returned component;

   (c) Node modification: when processing a given component, the content of the column `'action'` of the ruleset table of the related dependency rule is added to the node actions list.

3. **Rules Graph Making**: from the dependency rules table, and for a given ruleset, the corresponding graph – called the *rules graph* – is created. A node in that graph is a pair (s, t) where

ruleset	name	types	filter	action	depsfinder	dependson	comments
stop	coldoorOff	coldoor@hwmanager	ALL	bsmpower -a off %name	find_coldoorOff_dep %name	nodeOff	PowerOff nodes before a cold door
stop	nodeOff	compute@node|nfs@node	ALL	nodectrl poweroff %name	find_nodeoff_deps %name	nfsDown	Unmount cleanly and shutdown nfs properly before halting.
stop	nfsDown	nfsd@soft	ALL	@/etc/init.d/nfs stop	find_nfs_client %name	umountNFS	Stopping NFS daemons: take care of clients!
stop	umountNFS	umountNFS@soft	ALL	echo WARNING: NFS mounted!	NONE	NONE	Print a warning message for each client
start	coldoorStart	coldoor@hwmanager	ALL	bsmpower -a on %name	NONE	NONE	No dependencies
start	nodeOn	compute@node	%name =~ compute12	nodectrl poweron %name	find_nodeon_deps	coldoorStart	Power on cold door before nodes.
stopForce	daOffForce	da@disk_array	%name! ~ .*	da_admin poweroff %name	find_daOff_deps	ioServerDown	Unused thanks to Filter

Table 2: An example of a sequencer table.

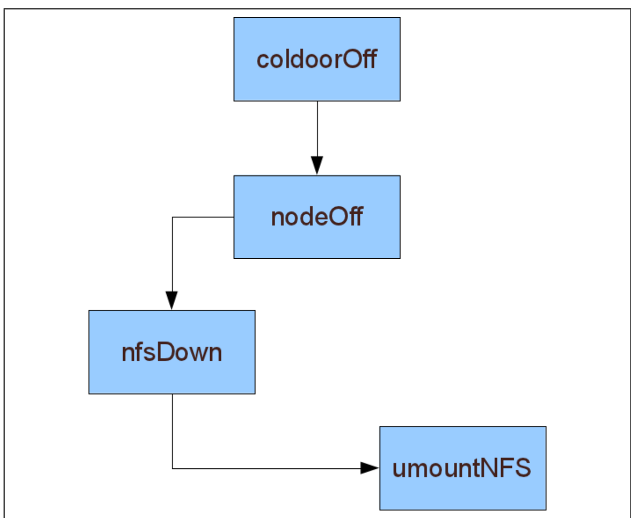


Figure 3.1: Rules Graph of the 'stop' ruleset defined in the sequencer table 2.

s is the rule symbolic name, and t is the component types defined by the 'types' column in the dependency rule table. This graph is used for the selection of a component in the components list to start the dependency graph update process. From table 2, the rules graph of the stop ruleset is shown figure 3.1. Note that cycles are possible in this graph. As an example, a PDU (related to a switch type in the sequencer table) that connects (power) an ethernet switch which itself connects (network) a PDU.

4. **Updating the Dependency Graph**: from each ids (resulting from the expansion of each initial components list), the corresponding rule in the given ruleset of the sequencer table should be found. For that purpose, a *potential root* is looked for in the ids set. A potential root is an id that *matches* one root[8] in the rules graph. A match between an id of the form 'name#type@category' and a rule 'R' occurs when type is in 'R.types' and when id has been *filtered in*. If such a match cannot be found, then, a new rules graph is derived from the preceding one by removing all roots and their related edges. Then, the root finding is done on that new graph, and so on recursively until either:
   - the rule graph is empty: in this case, the given components list cannot be started/stopped (entirely);
   - the rule graph is only made of cycles: any id can be used as the starting point;
   - a match occurs between 'id' and 'R': in this case, the dependency graph is updated from the application of rule 'R' to 'id'. Each time such an application is made, 'id' is removed from the initial id set.

[8]A node in the graph with no parent.

As an example, consider a rack cooled by a bullx cold door 'cd0' containing an NFS server 'nfs1' and a compute node 'c1'. Consider also another NFS server 'nfs2' that is not in the same rack. We also suppose that:

- 'c1' is client of both 'nfs1' and 'nfs2';
- 'nfs1' is client of 'nfs2';
- 'nfs2' is client of 'nfs1'[9].

Using table 2, and the component list: 'nfs1#nfsd@soft, cd0, nfs2', objectives are:

- power off 'c1' and 'nfs1' before 'cd0', because powering off a cold door requires that each equipement cooled are powered off first;
- stop NFS daemons on 'nfs1' because it is requested (this should be done before powering off 'nfs1');
- power off 'nfs2' because it is requested (but the NFS daemon will have to be stopped before);
- for each NFS client[10] a warning should be written before the actual stopping of used NFS server.

[9]Yes, it might seem strange here. This serves the purpose of our example.

[10]One might use the content of /var/lib/nfs/rmtab for an (inaccurate) list of NFS clients, the 'showmounts' command or any other means.

Figure 3.2: The initial dependency graph.

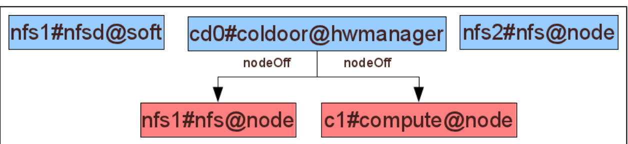


Figure 3.3: The dependendy graph after the call to the `'cd0#coldoor@hwmanager'` depsfinder.

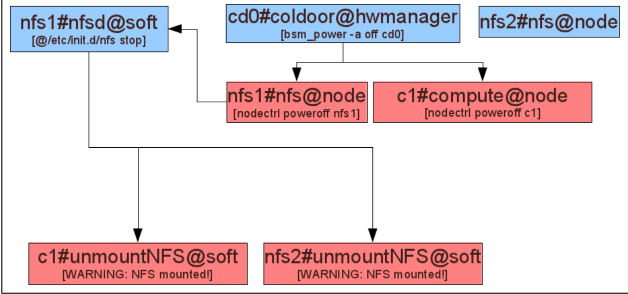


Figure 3.4: The dependency graph after the application of rule `'coldoorOff'` on `'cd0#coldoor@hwmanager'`.

With the table, the ruleset and the component list, the sequencer starts to expand the component list into an id set: `'nfs1#nfsd@soft, cd0#coldoor@hwmanager, nfs2#nfs@node'`. Then the dependency graph is initialized: each id in the set has a related node in the graph as shown in figure 3.2.

Then the sequencer looks for a *potential root* using the rules graph (shown on figure 3.1). A match exists between rule `'coldoorOff'` and `'cd0#coldooor@hwmanager'`. Therefore, the sequencer starts applying rule `'coldoorOff'` to `'cd0#coldooor@hwmanager'`. The depsfinder of the rule is called. For each id returned by the depsfinder[11], the graph is updated: a node with an empty action list is made and an edge from the current id to the dependency is created. Each returned id is added to the id set.

Then, for each dependency, the sequencer checks if a match exists with one of the rules defined in the `'dependson'` column, and for each match, the matching rule is applied recursively.

In our case, the cold door depsfinder returns every cooled component: `'nfs1#nfs@node'` and `'c1#compute@node'`. Therefore, the graph is updated as shown in figure 3.3. The `'coldoorOff'` rule defines a single dependency in its `'dependson'` column: `'nodeOff'`. Both components match, the rule is applied. The application of the rule `'nodeOff'` on `'c1#compute@node'` leads to the execution of the depsfinder which does not return anything. Therefore, the application of the rule ends by adding the action `'nodectrl poweroff %name'` to the related node in the dependency graph and by the removal of the related id from the id set.

This implies that a given rule is applied at most once on a given id.

The sequencer continues with the next dependency which is `'nfs1#nfs@node'`. The application of the rule `'nodeOff'` leads to the execution of the depsfinder which returns `'nfs1#nfsd@soft'`. This node is already in the graph (it is in the initial id set). Therefore, the dependency graph is just updated with a new edge. This id matches the dependency rule specified `'nfsDown'` and this last rule is applied on that id. The depsfinder on `'nfs1#nfsd@soft'` returns all known clients which are `'c1#unmountNFS@soft'` and `'nfs2#unmountNFS@soft'`.

Finally both dependencies match the rule `'umountNFS'` but its application does not lead to any new node in the dependency graph. However, the graph is updated so each node is mapped to its related action, recursively, up to the node the sequencer started with: `'cd0#coldoor@hwmanager'` as show on figure 3.4.

At that stage, the id set contains only the last element from the originial component list:

[11] Note that it is not required that depsfinder returns ids with a predefined category as soon as a match occurs in the sequencer table. Predefined categories are used to ease the mapping between a given component and a type. In a large cluster (or data-center), it may not be easy to determine what is the real type of a given component name.

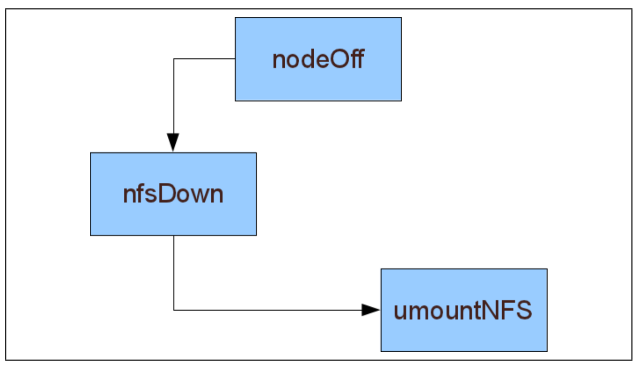


Figure 3.5: The rules graph with first level roots removed.

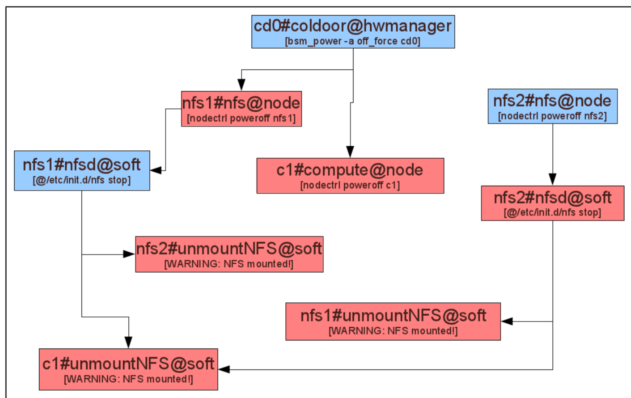


Figure 3.6: The final dependency graph.

`'nfs2#nfs@node'` (others were removed by preceding rule applications). Unfortunately that id does not match any root rule in the rules graph. Thus, the rules graph is (virtually) modified so roots are removed. This leaves us with the rules graph shown on figure 3.5.

From that graph, id `'nfs2#nfs@node'` matches root rule `'nodeOff'` which is therefore applied. The depsfinder returns `'nfs2#nfsd@soft'` which is new and therefore added in the dependency graph. The rule `'nfsDown'` is applied on that id (since a match occurs) giving us two dependencies `'c1#unmountNFS@soft'` and `'nfs1#unmountNFS@soft'`.

The algorithm ends after the mapping of those new ids with their related actions as shown in the final graph shown on figure 3.6.

Remember that a rule is never applied twice on a given id. Therefore, the action from rule `'unmountNFS'` which is `'echo WARNING: NFS mounted!'` on id `'c1#unmountNFS@soft'` is not added twice.

The sequencer displays this dependency graph in an XML format (using the open-source python-graph library available at `http://code.google.com/p/python-graph/`) on its standard output. This output can be given directly to the second stage of the sequencer. Note that contrary to the rules graph, the dependency graph should not contain a cycle (this will be detected by the next stage and refused as an input).

3.2 Instructions Sequence Maker (ISM)

The Instructions Sequence Maker (ISM) is the second stage of the sequencer. Its role is to transform a dependency graph into a set of instructions that can be given as an input to the third stage, the Instructions Sequence Executor (ISE).

A set of instructions is specified as an XML document, within an `<instructions>` XML tag. Three kind of instructions can be specified:

Action: defined by the `<action>` tag. It specifies the actual command that should be executed. Attributes are:

- `'id'`: Each action should be identified by a unique string. This attribute is mandatory. It is usually[12] of the form `'name#type@category!rule'`
- `'deps'`: a list of ids this action depends on (*explicit dependencies*). This attribute is optionnal. Default is the empty string.
- `'remote'`: the command should be executed using the current shell unless this attribute is set to `'true'`. In this case, an

[12]It is not required for the instructions sequence XML document to be created by the Instructions Sequence Maker. It may be created/modified by hand or by any other programs.

internal ssh connexion is made to execute the given command on each components described by the `'component_set'` attribute (see below). This attribute is optionnal. Default is `'false'`.

- '`component_set`': the set of components this action should be executed on in the following format: `'name[range]#type@category'`. This attribute is ignored by the ISE unless the remote attribute is set to `'true'`. This attribute is optionnal. Default is `'localhost#type@cat'`.

Sequence: defined by the `<seq>` tag. It specifies a set of instructions (hence, one of Action, Sequence or Parallel) that must be executed in the given order. This defines *implicit dependencies* between instructions as opposed to *explicit dependencies* defined by the `'deps'` attribute of an Action.

Parallel: defined by the `<par>` tag. It specifies a set of instructions (hence one of Action, Sequence or Parallel) that can be executed in any order. This explicitly defines that there is no dependency between each instruction. The ISE is free to execute them in parallel. Note that the ISE may or may not execute those instructions in parallel. This is not a requirement for the successful completion of a parallel instruction.

Transforming a dependency graph into an instructions sequence is straightforward if performance is not the main goal. A simple topological sort [6] on the input dependency graph returns a sequence of actions where constraints are respected.

For example, on our example where the final dependency graph computed by the DGM is given on figure 3.6, a topological sort[13] gives the sequence shown on sample 1.

This sequence is valid, but not efficient: it requires 9 sequential steps. This transformation algorithm is called `'seq'` in the sequencer and it can be selected. Three other algorithms are provided within the sequencer:

- `'par'`: this algorithm inserts each node in the dependency graph using a single parallel (`<par>` XML tag) instruction and explicit dependencies (`'deps'` attribute of the `<action>` XML tag). Such an algorithm is optimal in terms of performance, but it produces an instructions sequence file that is difficult to read by a human because of all those explicit dependencies.

- `'mixed'`: this algorithm inserts each leaf nodes in the dependency graph using a parallel instruction, then remove those leaf nodes from the dependency graph and starts again. Every such parallel instructions are included in a global sequence one (`<seq>` XML tag). This algorithm tends to execute set of actions by steps: all leaf nodes are executed in parallel. Once they have terminated, they are removed from the graph, and another batch of leaf nodes are executed in parallel up to the end.

- `'optimal'`: this algorithm produces an instructions sequence that is as efficient as the `'par'` algorithm but much more readable. It uses implicit dependencies as much as possible[14] using sequence instructions. This algorithm is selected by default.

Describing in details those algorithms with their advantages and constraints is beyond the scope of this paper.

3.3 Instructions Sequence Executor (ISE)

The Instructions Sequence Executor (ISE) is the last stage of the sequencer. It takes in input an instructions sequence as computed by the ISM or created/edited by hand or by any other means. It then runs the instructions specified taking into account:

[13] For a given directed acyclic graph, several valid topological sort outputs can be found.

[14] Our XML instructions sequence format can only express trees if implicit dependencies are used exclusively.

Sample 1 Result of the topological sort on the dependency graph given figure 3.6.

```
<instructions>
    <seq>
        <action id=''c1#unmountNFS@soft!unmountNFS''>echo WARNING: NFS mounted!</action>
        <action id=''nfs1#unmountNFS@soft!unmountNFS''>echo WARNING: NFS mounted!</action>
        <action id=''nfs2#unmountNFS@soft!unmountNFS''>echo WARNING: NFS mounted!</action>
        <action id=''nfs1#nfsd@node!nfsDown'' remote=''true'' component_set=''nfs1#nfsd@node''>
         /etc/init.d/nfsd stop
        </action>
        <action id=''nfs1#nfs@node!nodeOff''>nodectrl poweroff nfs1</action>
        <action id=''nfs2#nfsd@soft!nfsDown'' remote=''true'' component_set=''nfs2#nfs@node''>
         /etc/init.d/nfsd stop
        </action>
        <action id=''c1#compute@node''>nodectrl poweroff c1</action>
        <action id=''nfs2#nfs@node!nodeOff''>nodectrl poweroff nfs2</action>
        <action id=''cd0#coldoor@hwmanager!coldoorOff''>bsmpower -a off cd0</action>
    </seq>
</instructions>
```

- parallelism: actions that do not have dependencies between them might be executed in parallel. There is a customizable maximum limit on the number of actions that can be executed in parallel by the ISE. This helps limiting the load increase of the system due to a vast number of forks in a large cluster.
- dependencies: an action is not executed unless all its dependencies (explicit and implicit) have completed successfully. An executed action is considered successful in two cases:
  - its returned code is 0 (alias OK);
  - its returned code is 75 (alias WARNING also known as EX_TEMPFAIL in `sysexits.h`) and the option `'--Force'` has been given to the ISE.

The implementation of the ISE uses the ClusterShell [7] python library as the backend execution engine. Describing the implementation of the ISE is beyond the scope of this article.

4 Scalability Issues

Using the sequencer on a large cluster such as the Tera-100 can lead to several issues related to scalability.

4.1 Complexity

Several complexity in space and time can be identified for:

1. the production of the dependency graph produced by the DGM;
2. the production of the actions graph produced by the ISM;
3. the execution of the actions graph by the ISE.

This last complexity was our first concern due to our customer requirements. If theorical complexity has not (yet) been formally determined, the execution time of the sequencer for the production of the dependency graph of the Tera-100 is:

- 13 minutes 40 seconds for the start ruleset with 9216 nodes and 8941 edges in the dependency graph;
- 2 minutes 1 second for the stop ruleset with 9222 nodes and 13304 edges in the dependency graph.

The time taken by the ISM for the production of the actions graph from the previously computed dependency graph using the 'optimal' algorithm is:

- 4.998 seconds for the start with 4604 nodes and 8742 edges in the actions graph;
- 6.343 seconds for the stop with 4606 nodes and 9054 edges in the actions graph.

Finally, the time taken by the ISE to execute these actions graph is:

- 4 minutes 27 seconds for the start with:
  - 99.7% of actions executed (successfully or not);
  - 6.6% of actions that ends on error for various reasons;
  - 0.3% of actions not executed because some of their dependencies ends on error or was not executed.
- 9 minutes 23 seconds for the stop ruleset with:
  - 96.7% of actions executed (successfully or not);
  - 15.3% of actions that ends on error for various reasons;
  - 3.3% of actions not executed because some of their dependencies ends on error or was not executed

Explaining differences between those metrics is beyond the scope of this paper. However, from such results, the sequencer can be considered has quite scalable.

4.2 Mantainability

The maintenance of the various graph used by the sequencer:

- rules graph;
- the DGM produced dependency graph;
- the ISM produced actions graph XML file;

is also an issue on large systems. Identifying wrong dependencies in a flat file can be hard, especially with large graph represented with several thousands of lines.

The sequencer can exports those graph in the DOT format. It therefore delegates to specific graph tools, the identification of non trivial problems for maintenance purposes. For instance, rules graph are usually small and the standard `'dot'` command that comes within the graphviz [8] open-source standard product can be used for their vizualisation. This is fast and easy. For other much larger graph, however, specialized tools such as Tulip [9] might be used instead.

4.3 Usability

In the context of large systems, giving correct inputs to a tool, and getting back a usable output can be a big challenge in itself. In the case of the sequencer, inputs are the sequencer table and the components list.

For the maintenance of the table, the sequencer provides a management tool that helps adding, removing, updating, copying and even checksuming rules. For the components list, the sequencer uses what is called a `guesser` that given a simple component name fetches its type and category. This allows the end user to specify only the name of a component.

Apart from the output produced by the first two stages that have already been discussed in the previous section on maintanability, the last output of great interest for the end-user, is the ISE output. To increase further the usability of the sequencer, several features are provided:

Prefix Notation: each executed action output is prefixed by its id. When the ISE executes an action graph produced by previous stages, those ids contain various informations such as the type, the category, and the rulename this action comes from. This helps identifying which action produced which output (the bare mininum). Moreover, such an output can be passed to various filters such as grep or even gathering commands such as `'clubak'` from ClusterShell [7] or `'dshbak'` from pdsh [10]. In the case of

'clubak' command, the separator can be given as an option. As a side effect, this allows the end-user to group similar output by node, type or category.

Reporting: The ISE can produce various reports:

- 'model': each action with their dependencies (implicit and explicit) are shown; this is used to determine what the ISE will do before the actual execution (using a '--noexec' option);
- 'exec': each executed action is displayed along with various timing informations and the returned code;
- 'error': each executed action that exited with an error code is displayed along with their reversed dependencies (their parent in the dependency graph); this is used to know which action has not been executed because of a given error;
- 'unexec': each non executed action is displayed along with its missing dependencies — a dependency that exited with an error code and that prevented the action from being executed; this is used to know why a given action has not been executed.

5 Results

The sequencer has been designed for two main purposes:

1. Emergency Power Off: this is the reason of the three different independent stages;
2. Common management of resources in clusters (and data-centers): this is the main reason for the chaining mechanism.

Our first experiment with our tool shows that it is quite efficient. Our main target was the powering on/off of the whole Tera-100 system which leads to the execution of more than 4500 actions in less than 5 minutes for the start and in less than 10 minutes for the stop.

6 Related Works

Dependency graph makers exist in various products:

- Make [11], SCons [12], Ant [13] for example are used for the building of softwares; they focus on files rather that cluster components and are therefore not easily adaptable to our problem.
- Old System V init [14], BSD init [15], Gentoo init[16] and their successors Solaris SMF [17], Apple launchd [18], Ubuntu upstart [19] and Fedora systemd [20] are used during the boot of a system and for managing daemons. To our knowledge none of those products can be given a components list as an input so actions are executed based on it.

Solutions for starting/stopping a whole cluster are most of the time manual, described in a step-by-step chapter of the product documentation and hard wired. This is the case for example with the Sun/Oracle solution [21] (command 'cluster shutdown'). It is not clear whether all components in the cluster are taken into account (switches, cooling components, softwares, ...) and whether new components can be added to the shutdown process. IBM uses the open-source xcat [22] project and its 'rpower' command which does not provide dependencies between cluster components.

From a high level perspective, the sequencer, can be seen as a command dispatching framework similar to Fabric [23], Func [24] and Capistrano [25] for example. But the ability to deal with dependencies lacks in these products making them unsuitable for our initial problem.

The sequencer can also be seen as a workflow management system where the pair DGM/ISM acts as a workflow generator, and the ISE acts as a workflow executor. However, the sequencer has not been designed for human interactive actions. It does not deal for example with user access rights or tasks list for example. It is therefore much lighter than common user oriented workflow management systems such as YAWL [26], Bonita [27], Intalio|BPMS [28], jBPM [29] or Activiti [30] among others.

We finally found a single real product for which a comparison has some meaning: ControlTier [31].

ControlTier shares with the sequencer various features such as possible parallel execution of independent actions and failure handling. However, the main difference is in the way workflows are produced: they are dynamically computed in the sequencer case through dependency rules (depsfinder scripts) and the component input list whereas they are hard wired through configuration files in the case of ControlTier.

To our knowledge, our solution is the first one to address directly the problem of starting/stopping a whole cluster or a part of it, taking dependencies into considerations, still remaining integrated, efficient, customizable and robust in the case of failure. We suppose the main reason is that clusters are not designed to get started/stopped entirely. Long uptime is an objective! However automated tools ease the management of clusters, making start/stop procedure faster, reproducible, and reducing human errors to a minimum.

7 Conclusion and Future Works

The sequencer solution presented in this article is the first of its kind to our knowledge. It has been designed with EPO in mind. This is the reason for its 3 independent stages and for the incremental mode of execution. Still the sequencer provides the chaining feature making its use pertinent for small clusters or small part of a big one.

The sequencer fulfills our initial objectives:

- Predictive: the incremental mode allows a computed instructions sequence to be verified, modified and recorded before being run.
- Easy: executing a recorded instructions sequence requires a single command: `'clmsequencer < instructions.sequence'`
- Fast: the sequencer can execute independent instructions in parallel with a customizable upper limit.
- Smart: the order in which instructions are executed comes from a dependency graph computed from customizable dependency rules and a given cluster components list.
- Robust: failures are taken into account by the sequencer component by component.

Our solution is highly flexible in that most of its inner working is configurable such as:

- the dependency rules,
- the dependency fetching scripts,
- the action to be taken on each component,
- the dependency graph,
- the final instruction sets.

The sequencer has been validated on the whole Tera-100 system. A shutdown can be done in less than 10 minutes, and a power on takes less than 5 minutes (more than 4500 actions for both rulesets). The sequencer framework will be released under an open-source license soon. Several enhancements are planned for the end of this year including:

- smarter failure handling;
- live reporting/monitoring;
- performance improvement of dependency graph generation through caching;
- post-mortem reporting;
- replaying.

8 Aknowledgment

The author would like to thank Matthieu Pérotin for his helpful review, Marc Girard for his support during the development of the solution presented in this paper and the reviewers for their very clear comments and remarks.

References

[1] W. Turner, J. Seader, and K. Brill, "Industry standard tier classifications define site infrastructure performance," tech. rep., Uptime Institute, 2005. 1

[2] P. Grun, "Introduction to infiniband for end users," tech. rep., InfiniBand Trade Association, April 2010. `http://members.infinibandta.org/kwspub/Intro_to_IB_for_End_Users.pdf`. 1

[3] AMD, "Magic packet technology." White Paper, November 1995. Publication# 20213, Rev: A Amendment/0 `http://support.amd.com/us/Embedded_TechDocs/20213.pdf`. 2

[4] N. D. Intel, Hewlett-Packard, "Ipmi v2.0 rev. 1.0 specification markup for ipmi v2.0/v1.5 errata revision 4," 2009. `http://download.intel.com/design/servers/ipmi/IPMI2_0E4_Markup_061209.pdf`. 2

[5] S. R. International, "Snmp rfcs." `http://www.snmp.com/protocol/snmp_rfcs.shtml`. 2

[6] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, pp. 558–562, November 1962. 11

[7] "Clustershell opensource project," 2011. `http://sourceforge.net/apps/trac/clustershell`. 12, 13

[8] AT&T Labs Research and Contributors, "Graphviz." Web page, June 2011. `http://graphviz.org/`. 13

[9] D. Auber, "Tulip : A huge graph visualisation framework," in *Graph Drawing Softwares* (P. Mutzel and M. Jünger, eds.), Mathematics and Visualization, pp. 105–126, Springer-Verlag, 2003. 13

[10] J. Garlick, "pdsh: Parallel distributed shell." `http://sourceforge.net/projects/pdsh/`. 13

[11] Free Software Foundation, "GNU Make." Web page, July 2004. `http://www.gnu.org/software/make/`. 14

[12] The SCons Foundation, "SCons." Web page, June 2011. `http://www.scons.org/`. 14

[13] The Apache Software Foundation, "The Apache Ant Project." Web page, July 2004. `http://ant.apache.org/`. 14

[14] Novell, Inc (now SCO), *System V Interface Definition, Fourth Edition, Volume 2*, June 1995. `http://www.sco.com/developers/devspecs/vol2.pdf`. 14

[15] FreeBSD, *FreeBSD System Manager's Manual, init(8)*, Sept. 2005. `http://www.freebsd.org/cgi/man.cgi?query=init&sektion=8`. 14

[16] "Gentoo Initscripts," Mar. 2011. `http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=4`. 14

[17] R. Romack, "Service managemen facility (smf) in the solaris 10 operating system." Sun BluePrints OnLine, Feb. 2006. `http://www.linux.com/archive/feature/125977`. 14

[18] J. Wisenbaker, "launchd in depth." AFP548, July 2005. `http://www.afp548.com/article.php?story=20050620071558293`. 14

[19] M. Sobell, "Ubuntu's upstart event-based init daemon," Feb. 2008. `http://www.linux.com/archive/feature/125977`. 14

[20] L. Poettering, "Rethinking pid 1," Apr. 2010. `http://0pointer.de/blog/projects/systemd.html`. 14

[21] Oracle, *Oracle Solaris Cluster System Administration Guide*.
http://download.oracle.com/docs/cd/E18728_01/html/821-1257/ghfwr.html. 14

[22] L. Octavian, A. Brindeyev, D. E. Quintero, V. Sermakkani, R. Simon, and T. Struble, "xCAT 2 Guide for the CSM System Administrator." IBM Red Paper, 2008.
http://www.redbooks.ibm.com/redpapers/pdfs/redp4437.pdf. 14

[23] Jeff Forcier, "Fabric." Web page, June 2011.
http://fabfile.org/. 14

[24] MichaelDeHaan, AdrianLikins, and SethVidal, "Func: Fedora Unified Network Controller." Web page, June 2011.
https://fedorahosted.org/func/. 14

[25] Jamis Buck, "Capistrano." Web page, June 2011.
http://github.com/capistrano/capistrano/wiki/. 14

[26] M. Adams, S. Clemens, M. L. Rosa, and A. H. M. ter Hofstede, "Yawl: Power through patterns," in *BPM (Demos)*, 2009. 14

[27] M. V. Faura, "Bonita."
http://www.bonitasoft.com/. 14

[28] I. Ghalimi, "Intalio|bpms."
http://www.intalio.com/bpms. 14

[29] T. Baeyens, "Jbpm."
http://www.jboss.org/jbpm. 14

[30] T. Baeyens, "Activiti."
http://www.activity.org/. 14

[31] Alex Honor, "Control Tier." Web page, June 2011.
http://controltier.org/. 14

Automated Planning for Configuration Changes

Herry Herry
University of Edinburgh
Edinburgh, UK
h.herry@sms.ed.ac.uk

Paul Anderson
University of Edinburgh
Edinburgh, UK
dcspaul@ed.ac.uk

Gerhard Wickler
University of Edinburgh
Edinburgh, UK
g.wickler@ed.ac.uk

Abstract

This paper describes a prototype implementation of a configuration system which uses automated planning techniques to compute workflows between declarative states. The resulting workflows are executed using the popular combination of ControlTier and Puppet. This allows the tool to be used in unattended "autonomic" situations where manual workflow specification is not feasible. It also ensures that critical operational constraints are maintained throughout the execution of the workflow. We describe the background to the configuration and planning techniques, the architecture of the prototype, and show how the system deals with several examples of typical reconfiguration problems.

Keywords: configuration management, infrastructure, cloud computing, planning, IaaS

1 Introduction

The growing size and complexity of computing infrastructures has increased awareness of the need for good system configuration tools, and most sites now use some form of tool to manage their configurations. Furthermore, declarative specifications are now widely accepted as the most appropriate approach - the specification describes the "desired" state of the system, and the tool computes the necessary actions to move the system from its current state into this desired state. This has the advantage that the final *state* of the system is explicitly specified, and we can have some confidence that the state of the running system matches our requirements. Previous approaches were more error-prone because they involved specifying the *actions* (for example, using imperative scripts), and the final outcome would not always be obvious. With varying degrees of strictness, most of the currently popular tools take a broadly declarative approach - for example, Puppet [16] , Cfengine [3], BCFG [4] and LCFG [1].

However, none of the above tools make any guarantees about the order of the changes involved in implementing a configuration change. When creating a new service, this is not normally an issue - we specify the requirements and the tool makes all the necessary changes (in some random order). When the tool has finished, we have a running system to our specification. However, if we are making configuration changes to an existing system, we may well care about the state of the configuration during the change; for example, if we want to make a transition from using one server, to using a different one, then we probably want to start the new server, and transfer the clients before shutting down the old one.

Such transitions are often performed manually - the administrator will work out a number of intermediate stages (server B started, clients all using server B, server A stopped), and check that each state has been achieved before presenting the tool with the next state. However, this is both time consuming, error prone, and not suitable for unattended use - for example where we want to make a configuration change "autonomically" in response to some failure or change in load.

One approach to this problem has been the use of manual workflow tools. These allow workflows such as the previous example to be captured and stored for automatic use - a particular workflow can then be invoked and the tool will take care of scheduling the separate stages in the given order. ControlTier [5] and IBM Tivoli Provisioning Manager [12] are examples which provide this kind of capability. However, this still requires that the workflows are computed manually. Even in a small system, a very large number of workflows can be required to cater for every eventuality - for example, moving from every possible failed state into a working state. And choosing an appropriate workflow to suit a particular goal state is not always obvious - indeed such manual workflows are conceptually similar to the imperative scripts which are no longer popular because of their unreliability.

An alternative approach is to make use of automated

planning technology to generate workflows "on the fly". This allows us to specify the current and goal states, together with a set of constraints on the intermediate stages - for example, all clients must always point at a working server. The intermediate states are then computed automatically, and these can then be presented in order to the configuration tool to effect a smooth transition.

This paper describes an experimental system which applies established AI planning tools to automatically generate workflows between declarative system states. The resulting intermediate states are implementable in Puppet and can be scheduled by ControlTier to produce a fully-automated system. We start (section 2) with a full "walk through" of a simple example, based on the server-transition problem described above. Section 3 then covers the background in more detail, including system configuration and automated planning technology. Section 4 describes the prototype system, section 5 presents some more complex examples, section 6 concludes with a discussion of some of the problems, and section 7 presents possible future directions.

2 An Example

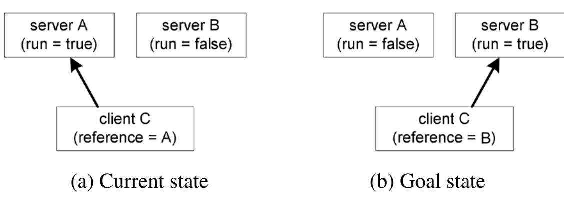
Assume we have a system consisting of two servers A and B, and one client C. Figure 1a shows the *current state*:

1. A.run = *true* (A is running),
2. B.run = *false* (B is stopped),
3. C.server = A (C is using a service of A).

The administrator aims to change the configuration to the *goal state* shown in figure 1b i.e.:

1. A.run = *false* (A is stopped),
2. B.run = *true* (B is running),
3. C.server = B (C is using a service of B).

Since C depends on the server's service, the changes must be implemented under a particular constraint i.e. C must always reference a running server.



(a) Current state (b) Goal state

Figure 1: States of the system.

If we use any declarative tool to implement these changes, then there are six possible sequences of states that could occur i.e.:

1. A.run = *false*, C.server = B, B.run = *true*;
2. C.server = B, A.run = *false*, B.run = *true*;
3. B.run = *true*, A.run = *false*, C.server = B;
4. A.run = *false*, B.run = *true*, C.server = B;
5. C.server = B, B.run = *true*, A.run = *false*;
6. **B.run = *true*, C.server = B, A.run = *false*.**

Any of these sequences could appear in practice because the declarative tools implement the changes by executing the actions in an essentially indeterminate order. Unfortunately, only one of these sequences (#6), satisfies the required constraint while others do not. Hence, a declarative tool is highly likely to produce change sequence which leaves the system inoperative for a period of time during the change.

To address the problem, the automated planning technique used in our prototype creates the workflow automatically, based on the given goal states and the available actions. The prototype will generate a workflow which consists of a sequence of actions that satisfies an ordering constraint. Each action has *preconditions* which are constraints that have to be satisfied before executing the action, and *effects* which are states that will be attained after executing the action.

The prototype has the following actions pre-defined in the actions database:

1.
```
start-server
 parameters: <server>
 preconditions: <server>.run = false
 effects: <server>.run = true
```
2.
```
stop-server
 parameters: <server>
 preconditions:
  <server>.run = true
  (forall <client>
    <client>.server != <server>)
 effects: <server>.run = false
```
3.
```
change-reference
 parameters: <server1> <server2> <client>
 preconditions:
  <client>.server = <server1>
  <server2>.run = true
  <client>.server != <server2>
 effects: <client>.server = <server2>
```

To generate the workflow, the administrator (or some autonomic system) simply needs to declare the goal states:

1. C.server = B

2. A.run = *false*

Based on the above goal states and the available actions, our prototype generates the following ControlTier workflow:

```
<command name="config_changes"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="1">
    <command name="start-server_B"/>
    <command name="reset-reference_A_B_C"/>
    <command name="stop-server_A"/>
  </workflow>
</command>

<command name="start-server_B" description=""
 command-type="Command" is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-server.pp B
   </argument-string>
</command>

<command name="change-reference_A_B_C"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-reference.pp A B C
   </argument-string>
</command>

<command name="stop-server_A" description=""
 command-type="Command" is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-server.pp A
   </argument-string>
</command>
```

Submitting this workflow to ControlTier implements the configuration change using the valid sequence of actions (#6).

This example shows that the prototype is able to eliminate the sequencing problem that exists in declarative tools. Moreover, the prototype also simplifies the configuration tasks since it only requires the administrator to declare the goal states, and not the explicit workflow.

3 Background

This section summarises the approaches to system configuration discussed in the introduction, and surveys some background work on automated planning techniques. It concludes with a discussion of some related work in applying planning techniques to the configuration problem.

3.1 System Configuration

As noted in the introduction, approaches to system configuration have mostly evolved via the the following stages:

- **Manual configuration** - the administrator manually computes the actions necessary to change from one configuration to another, and then manually executes the commands necessary to implement this. Clearly, this is error prone, time-consuming, and it is difficult to prove reliably that a given sequence of changes will result in the required configuration under all circumstances.

- **Scripted changes** - similar to the previous approach, except that the sequence of changes is captured in an imperative script, allowing it to be executed multiple times, on different systems. This clearly makes it easier to deal with large numbers of systems, and until comparatively recently, this was probably the most common approach to configuration for many people. However, scripting still suffers from most of the problems of the manual approach. In particular, there is a tendency to blindly apply scripts to situations which do not meet the necessary preconditions, and the outcome can be very unpredictable.

- **Declarative tools** - currently, the most common approach in practice is probably to use a tool which allows a declarative specification of the desired state. The tool will then compute and implement the necessary changes (in an essentially indeterminate order). This guarantees that the resulting configuration matches the required specification, regardless of the starting point. As noted in the introduction, typical tools include Puppet [16], Cfengine [3], BCFG [4] and LCFG [1].

- **Fixed workflow orchestration** - in many cases, it is now essential to be able to perform sequences on configuration changes automatically, and/or unattanded, and use of fixed workflow tools is becoming more common. As noted in the introduction, ControlTier [5] and IBM Tivoli Provisioning Manager [12] are typical examples.

3.2 Automated Planning

Automated planning techniques generate a plan (workflow) automatically by computing a sequence of actions which will transition a system from some *initial state* to some required *goal state*. Each action has *preconditions* which are constraints that have to be satisfied before executing the action, and *effects* which are conditions that will be true after executing the action[1].

[1] Formally, a planning problem can be defined as $P = (\Sigma, s_i, S_g)$, where $\Sigma = (S, A, \gamma)$ is a state transition system, $s_i \in S$ is the initial state, and $S_g \subset S$ is a set of goal states, A is a set of actions, γ is a state transition function, find a sequence of action $\langle a_1, a_2, \ldots, a_k \rangle$ cor-

Practical implementations of automated planners use search algorithms to find an appropriate sequence of actions. There are several approaches to improving the efficiency of a simple brute-force search:

- **State-space planning** uses a subset of the state space as the search space where nodes correspond to the world states, arcs correspond to the state transitions and a path in the search space corresponds to the plan. The algorithms try to find a plan that satisfies the goal from the current state using particular heuristics to minimize the computing time. Metric-FF [10] and SGPlan [11] are examples of planners that use this approach.

- **Plan-space planning** uses the plan space as the search space where nodes are partially specified plans and arcs are the plan refinement operations intended to further complete a partial plan. The algorithms try to eliminate all the flaws in the initial partial plan which is either an unsatisfied sub-goal or a threat. The final plan will bring the system from the initial to the goal state. Planners that use this approach include UCPOP [15] and VHPOP [20].

- **Planning-Graph** uses a graph structure where nodes correspond to world state propositions and actions, and arcs correspond to preconditions and effects of actions. The algorithms expand the graph from the initial state until reaching the last layer that contains all goals which must not be mutually exclusive. The solution (plan) can be found by applying a backward-search algorithm from the last until reaching the first proposition layer. Graphplan planner [2] and LPG [8] are examples of planners that use this approach.

- **Hierarchical Task Network (HTN)** planning uses algorithms that decompose the given tasks using pre-defined methods until it reaches a set of primitive tasks and no non-primitive tasks remain. The tasks are organized as a collection called a task network which consists of a set of tasks and a set of constraints. O-Plan [19] is an example of planner that use this approach.

3.3 Related Works

There has been some previous works on the use of automated planning techniques for sequencing configuration changes in computing infrastructures. For example:

Keller et al. [13] introduced CHAMPS which translates the requested operations into a set of imperative tasks and organizes them as a workflow to satisfy the constraints as well as maximize the high degree of parallelism. However CHAMPS does not reason about the current state of target system as well as the preconditions and effects of each task which could lead to an unsound plan.

In [6], an object modelling language is used to specify the goal states and the operational capabilities of the configured components. The model is mapped in Planning Domain Definition Language (PDDL) [7] and given as the input to a POP (partial-order planning) planner which generates the workflow. Hagen [9] models the component life-cycle using CIM (Common Information Model) objects which are stored in a Configuration Management Database (CMDB). Based on the defined goal states, the state-space planner called LPS (Logical Planning Stat) then directly manipulates the objects in the CMDB to generate the workflow.

Both approaches demonstrate the viability of automated planning techniques for changes to the configuration of a computing infrastructure. They also provide very flexible solutions. However the modelling and the specifications are comparatively complex, and it is not clear how these might be exposed to end-users in a practically useful way. Levanti [14] provides a promising approach to simplifying the interface to the planner – the user is presented with a set of tags which hide much of the configuration details (states and operations). This enables the user to define and refine the goal state by iteratively selecting one or more tags. The workflow is generated by mapping the selected tags in SPPL (Stream Processing Planning LanguageL) [17] as the input for the SPPL planner [18].

We are not aware of any other work which meets our specific aims of using a standard planner to create a system which interfaces easily with common system configuration tools.

4 Prototype

We have developed a prototype implementation which combines an automated planner, together with ControlTier and Puppet to generate and execute plans for configuration changes. This prototype is definitely not (yet) a production-quality tool. However, our main aim has been to prove that the concepts would be applicable to a real environment, so the tool uses production-quality components, and is capable of generating practical workflows from specifications of realistic problems.

As illustrated in figure 2, the prototype consists of four main components i.e. actions database, translator, planner, and mapper. More details of the architectures' component are described as follows (each number represents the component's number in figure 2):

responding to a sequence of state transitions $\langle s_i, s_1, \ldots, s_k \rangle$ such that $s_1 = \gamma(s_i, a_1), s_2 = \gamma(s_1, a_2), \ldots, s_k = \gamma(s_{k-1}, a_k)$, and $s_k \in S_g$.

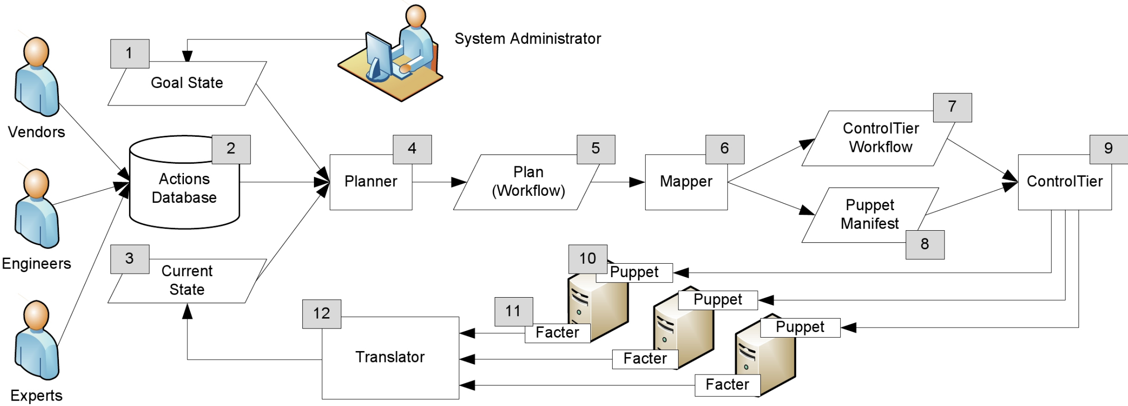


Figure 2: System Architecture of The Prototype

- The actions database (2) holds a library of actions, together with their required preconditions and effects. The actions can be written by anyone such as third-party software vendors, in-house software engineers, system administrators, or other specialists.
- A tool called facter (11) is used to acquire the current state (3) of the system. The outputs are aggregated by a translator (12) and then mapped into PDDL.
- The administrator specifies a declarative goal state (1) which is then mapped into PDDL.
- The planner (4) generates a plan (5) that will implement the new specification on the target system. This is based on the available predefined actions (from the actions database), the current state (from facter) and the goal state (supplied by the administrator).
- The mapper (6) uses the plan (5) to generate a ControlTier workflow-command (7) which consists of a set of other workflow-commands or primitive-commands. For each primitive-command, the mapper generates a puppet manifest file (8) to implement the action associated with the plan.
- ControlTier (9) manages the execution of the workflow by sending the puppet manifest file to the appropriate target node and requesting Puppet (10) to implement it.

In the latest prototype, we use LPG [8] as the planner. The main reason is that LPG can generate a plan (workflow) where all actions in each stage are mutually exclusive. This is an advantage because actions of each stage can be invoked in parallel by declarative tools to reduce the execution time. However, since we use the PDDL version 2.1 as the standard input for the planner, the prototype can utilize any available automated planner that supports it.

Currently, besides the translator, all parts have been implemented in Ruby and C. The implementation of the translator is straightforward i.e. translating the facts that are aggregated by the facter to a set of propositions in PDDL. For the actions database, all available actions are currently stored as individual files. If the number of actions were to become large, we could use a more structured database to improve storing and querying performance. For configuring other equipment (e.g. a router), we could employ a proxy server as a bridge to communicate with the target equipment in order to implement the new specification and acquire its current state.

In the process of generating the plan, the planner may use generic and domain-specific actions. A generic action, which is called as a “configuration pattern”, is a reusable action which is applicable on any configuration problem. Whilst a domain-specific action is an action which is applicable to particular configuration problem. We will show the examples of these types of action in the experiments section.

An error could occur during the implementation of any part of the plan. For example, the “change-reference” action cannot be executed if the target server is broken. To address this problem, the prototype could be set to identify the error from the execution log and perform the re-planning process to compute an alternative plan in order to attain the same goal state. If the alternative plan exists, it will then be implemented on the target system. Otherwise, the prototype could ask the administrator to modify the goal state.

The prototype could also have a self-healing capability simply by evaluating the current and the goal state periodically. It will then generate and execute a plan for correcting any drift in the configuration of the system.

5 Experiments

5.1 Web Services

In the first experiment, we reconfigure a system consisting of two web services WS-A and WS-B, a client PC, and a firewall FW. Currently, PC is using a web service provided by WS-A through port 8080 of FW and WS-B is stopped. As shown in figure 3a, the system's *current state* is:

1. WS-A.run = *true*
2. WS-A.enable_firewall = *true*
3. WS-A.FW.port = 8080
4. WS-B.run = *false*
5. WS-B.enable_firewall = *true*
6. PC.service = WS-A
7. FW.ports(8080).open = *true*
8. FW.ports(9090).open = *false*

The administrator aims to shutdown WS-A for maintenance and redirect PC's reference to WS-B. This will change the configuration to the *goal state* shown in figure 3b which can be specified declaratively as follows:

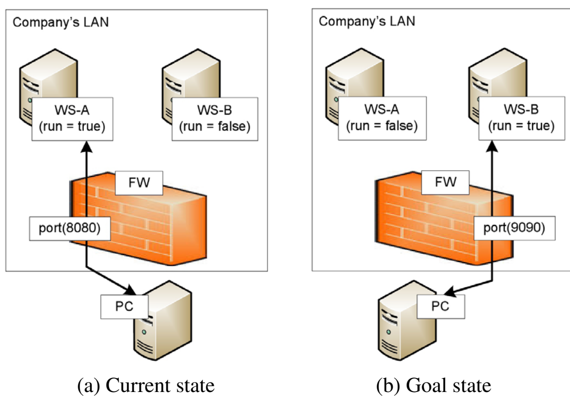
1. WS-A.run = *false*
2. PC.service = WS-B
3. WS-B.FW.port = 9090
4. FW.ports(8080).open = *false*

In addition, the administrator must satisfy the following constraints in the implementation of the changes:

1. The PC depends on the web service, thus it must always reference to a running web service.
2. Any unused port of F must be closed to minimize the vulnerability of the system.

To enable the planner to generate the right workflow, the first constraint is put in the preconditions of action *stop-service*. And the second one is declaratively specified in the goal state (#4). The following applicable actions are available in the actions database:

1.
```
start-service
 parameters: <service> <vm>
 preconditions:
  <service>.run = false
  <vm>.has = <service>
  <vm>.run = true
 effects: <service>.run = true
```



(a) Current state (b) Goal state

Figure 3: The states of the web services system.

2.
```
stop-service
 parameters: <service>
 preconditions:
  <service>.run = true
  (forall (<client>)
    <client>.service != <service>)
 effects: <service>.run = false
```

3.
```
open-fport
 parameters: <firewall> <port>
 preconditions:
  <firewall>.<port>.open = false
 effects:
  <firewall>.<port>.open = true
```

4.
```
close-fport
 parameters: <firewall> <port>
 preconditions:
  <firewall>.<port>.open = true
  (forall (<service>)
    <service>.<firewall>.port = <port>)
 effects:
  <firewall>.<port>.open = false
```

5.
```
assign-fport
 parameters: <service1> <firewall> <port>
 preconditions:
  <firewall>.<port>.open = true
  <service1>.enable_firewall = true
  <service1>.<firewall>.port != <port>
  (forall (<service2>)
    <service2>.<firewall>.port != <port>)
 effects:
  <service1>.<firewall>.port = <port>
```

6.
```
unassign-port
 parameters: <service> <firewall> <port>
 preconditions:
  <service>.<firewall>.port = <port>
  (forall (<client>)
    <client>.service != <service>)
 effects:
  <server>.<firewall>.port != <port>
```

7.
```
change-ref-fport
 parameters: <service1> <service2>
  <client> <firewall> <port>
 preconditions:
```

```
   <client>.service = <service1>
   <client>.service != <service2>
   <service2>.run = true
   <service2>.<firewall>.port = <port>
  effects:
   <client>.service != <service1>
   <client>.service = <service2>
```

By using information of the current and goal state with the application actions in the actions database, the prototype generated the following ControlTier workflow:

```
<command name="config_changes"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="1">
   <command name="sub-workflow-1"/>
   <command name="assign-fport_WS-B_FW_P9090"/>
   <command name=
    "change-ref-fport_WS-A_WS-B_PC_FW_P9090"/>
   <command name="sub-workflow-2"/>
   <command name="close-fport_FW_P8080"/>
  </workflow>
</command>

<command name="sub-workflow-1"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="start-service_WS-B_VM-B"/>
   <command name="open-fport_FW_P9090"/>
  </workflow>
</command>

<command name="sub-workflow-2"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="stop-service_WS-A"/>
   <command name="unassign-fport_WS-A_FW_P8080"/>
  </workflow>
</command>
```

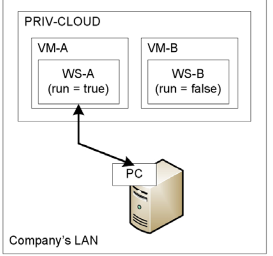
The prototype also generated the primitive ControlTier commands as shown in appendix B.

The generated workflow is a partial-order workflow which consists of one main workflow-command (*config_changes*) and two sub-workflow-commands (*sub-workflow-1* and *sub-workflow-2*). *config_changes* is set to be executed by one thread to enforce the ordering constraint, i.e. a command must be invoked after the previous one has finished successfully. On the other hand, each sub-workflow-commands is set to be executed by two threads[2]to enable the parallel execution. This is possible since all commands of the sub-workflow-command are mutually exclusive.

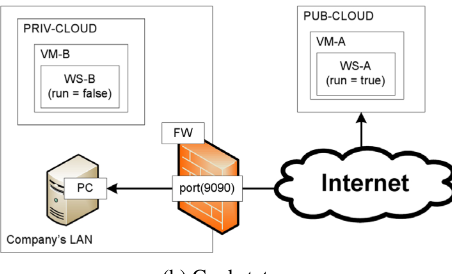
To implement the changes, the workflow was submitted by the mapper to ControlTier which coordinated the execution of each commands. Puppet then used the appropriate manifest file to acheive the desired state.

[2]The number of threads is the same as the number of primitive commands.

5.2 Cloud Burst



(a) Current state



(b) Goal state

Figure 4: The states of the company's system before and after the cloud-burst scenario.

In the second experiment, we simulated the cloud-burst scenario on the computing infrastructure. In this scenario, an organization must dynamically deploy its software application from its limited internal computing resources to the public cloud in order to address a spike in demand.

We assumed a company has a private cloud infrastructure which runs various services to serve its 24-hours operations. One of them, WS-A which is running on virtual machine VM-A, is the most important web service since it processes all financial transaction from company's branch offices. Thus, the administrator has prepared a backup web service, WS-B which is installed on virtual machine VM-B, in case there is a failure on WS-A.

Unfortunately, due to the limited resource of the physical machines, the company's private cloud infrastructure is not capable of serving the spikes in demand which usually happens on the last three days of each month. Therefore, before the spike's period, the administrator plans to migrate WS-A temporarily to the public cloud to minimize its response time.

The migration of WS-A from the private to the public cloud is not an easy task since the administrator must satisfy the following constraints:

1. During the migration process, the service must always available for 24-hours a day without any down-time.

2. The company's firewall must be reconfigured to allow the LAN PCs to have connection with the server on public cloud.

3. The web service application cannot be installed on any other machines due to the limitation of the license.

Based on the above scenario, the current state of the system is illustrated in figure 4a which can be specified declaratively as:

1. VM-A.cloud = PRIV-CLOUD
2. VM-A.run = true
3. WS-A.on = VM-A
4. WS-A.run = true
5. PC.service = WS-A
6. VM-B.cloud = PRIV-CLOUD
7. VM-B.run = false
8. WS-B.on = VM-B
9. WS-B.run = false

Where *PRIV-CLOUD* and *PUB-CLOUD* are the private and public cloud infrastructure respectively.

To enable the cloud-burst, the system needs to achieve the goal state as illustrated in figure 4b. Therefore, the administrator can reconfigure the system using our prototype by declaring the goal state as:

1. VM-A.cloud = PUB-CLOUD
2. WS-A.FW.port = 8080
3. PC.service = WS-A
4. VM-B.cloud = PRIV-CLOUD
5. VM-B.run = false

Where FW is the name of the company's firewall.

Fortunately, to generate the workflow, we only need to add five actions to the actions database since the planner can reuse the actions from the previous examples. The five new actions are:

1. `start-vm`
```
parameters: <vm> <cloud>
preconditions:
 <cloud>.has = <vm>
 <vm>.run = false
effects:
 <vm>.run = true
```

2. `stop-vm`
```
parameters: <vm>
preconditions:
 <vm>.run = true
 (forall <service>
   if <vm>.has = <service>
   then <service>.run = false)
effects:
 <vm>.run = false
```

3. `change-ref`
```
parameters: <service-1> <service-2> <client>
preconditions:
 <client>.service = <service-1>
 <client>.service != <service-2>
 <service-2>.run = true
 <service-2>.enable_firewall = false
effects:
 <client>.service != <service-1>
 <client>.service = <service-2>
```

4. `migrate`
```
parameters: <vm> <cloud-1> <cloud-2>
preconditions:
 <vm>.run = false
 <cloud-1>.has = <vm>
 <cloud-2>.is_public = true
effects:
 !(<cloud-1>.has = <vm>)
 <cloud-2>.has = <vm>
```

5. `set-need-firewall`
```
parameters: <service>
preconditions:
 (forall (<firewall> <port>)
   <service>.<firewall>.port != <port>)
 (forall (<vm> <cloud>)
   if (<vm>.has = <service>
       and <cloud>.has = <vm>)
   then <cloud>.is_public = true)
effects:
 <service>.enable_firewall = true
```

Some of these reusable actions, such as *stop-service* and *start-service*, typically occur in many different situations and form a set of generic *patterns*.

After processing the information, the planner will give its output to the mapper which generated the following ControlTier workflows:

```
<command name="config_changes"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="1">
   <command name="sub-workflow-1"/>
   <command name="start-service_WS-B_VM-B"/>
   <command name="change-ref_WS-A_WS-B_PC"/>
   <command name="stop-service_WS-A"/>
   <command name="stop-vm_VM-A"/>
   <command name=
    "migrate_VM-A_PRIV-CLOUD_PUB-CLOUD"/>
   <command name="sub-workflow-2"/>
   <command name="sub-workflow-3"/>
   <command name=
    "change-ref-fport_WS-B_WS-A_PC_FW_P8080"/>
   <command name="stop-service_WS-B"/>
```

```
    <command name="stop-vm_VM-B"/>
  </workflow>
</command>

<command name="sub-workflow-1"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="open-fport_FW_P8080"/>
   <command name="start-vm_VM-B_PRIV-CLOUD"/>
  </workflow>
</command>

<command name="sub-workflow-2"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="set-need-firewall_WS-A"/>
   <command name="start-vm_VM-A_PUB-CLOUD"/>
  </workflow>
</command>

<command name="sub-workflow-3"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="assign-fport_WS-A_FW_P8080"/>
   <command name="start-service_WS-A_VM-A"/>
  </workflow>
</command>
```

The prototype also generated the primitive ControlTier commands as shown in appendix C.

The planner generated the partial-order plan (workflow) which has three sub-workflows. Each sub-workflow has a set of commands that can be run in parallel due to their a mutual exclusive property. Submission the workflows to ControlTier implemented the new configuration specification that enable WS-A servicing more clients than before.

If the administrator would like to stop using the public cloud, WS-A can be migrated back to the private cloud by easily changing the goal state of the system. The prototype will generate and execute automatically the workflow to implement the new specification. In this case, we can also have a full autonomic configuration tool by replacing the administrator with an autonomic agent which will automatically trigger the migration of the system from private to the public cloud or vice versa based on the demands.

6 Conclusions

This work has clearly demonstrated the advantages of automated planning for system reconfiguration – workflows can be automatically generated (providing that a solution exists) between any two declarative states, enabling unattended, autonomic reconfiguration for failure recovery or other reasons. The generated workflows are guaranteed (by design) to achieve the desired target state, at the same time as preserving any necessary properties of the system during the reconfiguration. We have also shown that it is possible to build a practical tool which generates workflows automatically, and uses existing production-quality tools for the deployment (as well as the planning).

However, we suspect that the usability of such systems will be a major challenge – firstly, languages and interfaces are required to enable working administrators to easily translate their requirements and specifications into a form that is usable by the planners. Secondly, administrators need to have confidence that the system will behave in a predictable way – planners are very good at exploiting a lack of precision in the specification to find very "creative" and unexpected solutions! The human interaction aspects of this problem are something which would benefit from future work.

Error recovery is also a very important area. Reconfigurations often occur in precisely those situations where the system itself is unreliable – for example, during network and components failures, or system overload. Plans are likely to fail at some intermediate stage, or a centralised planner may become disconnected and lose track of the current state of an executing plan.

7 Future Work

We are currently interested in investigating more distributed, and localised approaches to automated planning for configuration changes. This will allow more autonomy for individual components (thus improving the resilience) and break the planning problem into a hierarchy of problems which are easier to understand and predict.

We believe that our implementation is much closer than previous work to providing a practical solution for system administrators who are familiar with current configuration tools such as Puppet. However, most administrators would still be unhappy to allow significant changes to their infrastructure by a completely automated system - the chances of unexpected and inappropriate solutions are still too high. We believe that there is considerable scope here for further work on appropriate languages and interfaces - perhaps involving *mixed initiative* solutions which combine automated planning with human guidance, and automated explanations of proposed solutions.

8 Acknowledgments

The authors like to thank Andrew Farrell from HP Labs Bristol for his valuable contributions. This research is fully supported by a grant from 2010 HP Labs Innovation Research Program Award.

References

[1] ANDERSON, P., AND SCOBIE, A. LCFG: The next generation. In *UKUUG Winter Conference* (2002).

[2] BLUM, A., AND FURST, M. Fast Planning through Planning Graph Analysis. *Artificial Intelligence 90* (1997), 281–300.

[3] CFENGINE AS. Cfengine - Automatic Server Lifecycle Management, 2011.

[4] DESAI, N., LUSK, A., BRADSHAW, R., AND EVARD, R. BCFG: A Configuration Management Tool for Heterogeneous Environments. In *Proceedings of IEEE International Conference on Cluster Computing* (2003), IEEE Computer Society.

[5] DTO SOLUTIONS. ControlTier, 2011.

[6] EL MAGHRAOUI, K., MEGHRANJANI, A., EILAM, T., KALANTAR, M., AND KONSTANTINOU, A. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware* (2006), pp. 404–423.

[7] FOX, M., AND LONG, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research 20*, 1 (2003), 61–124.

[8] GEREVINI, A., AND SERINA, I. LPG: A planner based on local search for planning graphs with action costs. In *Proceedings of the Sixth Internatinal Conference on AI Planning and Scheduling* (2002), pp. 12–22.

[9] HAGEN, S., AND KEMPER, A. Model-Based Planning for State-Related Changes to Infrastructure and Software as a Service Instances in Large Data Centers. In *2010 IEEE 3rd International Conference on Cloud Computing* (2010), pp. 11–18.

[10] HOFFMANN, J. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research 20*, 20 (2003), 291–341.

[11] HSU, C., WAH, B., HUANG, R., AND CHEN, Y. Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India* (2007).

[12] IBM CORP. Integrated Service Management software, IBM Tivoli, 2011.

[13] KELLER, A., HELLERSTEIN, J., WOLF, J., WU, K., AND KRISHNAN, V. The CHAMPS system: Change management with planning and scheduling. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP* (2004), vol. 1, pp. 395–408.

[14] LEVANTI, K., AND RANGANATHAN, A. Planning-based configuration and management of distributed systems. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on* (2009), pp. 65–72.

[15] PENBERTHY, J., AND WELD, D. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning* (1992), pp. 103–114.

[16] PUPPET LABS. Puppet, 2011.

[17] RIABOV, A., AND LIU, Z. Planning for stream processing systems. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3* (2005), vol. 20, pp. 1205–1210.

[18] RIABOV, A., AND LIU, Z. Scalable planning for distributed stream processing systems. In *Proceedings of ICAPS* (2006).

[19] TATE, A., DALTON, J., AND LEVINE, J. O-Plan: A web-based AI planning agent. In *Proceedings of the National Conference on Artificial Intelligence* (2000), pp. 1131–1132.

[20] YOUNES, H., AND SIMMONS, R. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research 20*, 1 (2003), 405–430.

A The Flow-Chart of The Workflows

Figure 5a and 5b illustrate the flow-charts of the generated workflows of web services and cloud burst examples. Each actions are associated with a ControlTier command as follows:

- a_1: `start-service_WS-B_VM-B`
- a_2: `open-fport_FW_P9090`
- a_3: `assign-fport_WS-B_FW_P9090`
- a_4: `change-ref-fport_WS-A_WS-B_PC_FW_P9090`
- a_5: `stop-service_WS-A`
- a_6: `unassign-fport_WS-A_FW_P8080`
- a_7: `close-fport_FW_P8080`
- b_1: `open-fport_FW_P8080`
- b_2: `start-vm_VM-B_PRIV-CLOUD`
- b_3: `start-service_WS-B_VM-B`
- b_4: `change-ref_WS-A_WS-B_PC`
- b_5: `stop-service_WS-A`
- b_6: `stop-vm_VM-A`
- b_7: `migrate_VM-A_PRIV-CLOUD_PUB-CLOUD`
- b_8: `set-need-firewall_WS-A`
- b_9: `start-vm_VM-A_PUB-CLOUD`
- b_{10}: `assign-fport_WS-A_FW_P8080`
- b_{11}: `start-service_WS-A_VM-A`
- b_{12}: `change-ref-fport_WS-B_WS-A_PC_FW_P8080`
- b_{13}: `stop-service_WS-B`
- b_{14}: `stop-vm_VM-B`

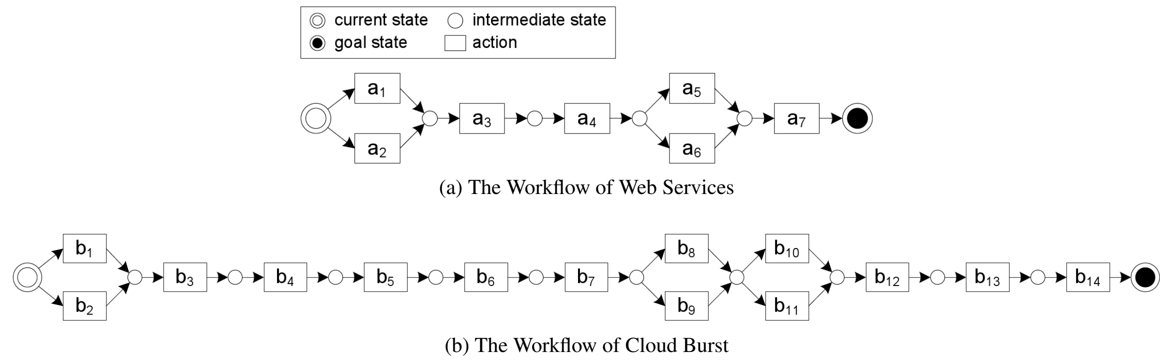


Figure 5: The flow-chart of the workflows.

B Primitive ControlTier Commands of Web Services

```
<command
 name="start-service_WS-B_VM-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-B VM-B
   </argument-string>
</command>

<command name="open-fport_FW_P9090"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-open-fport.pp FW 9090
   </argument-string>
</command>

<command name="assign-fport_WS-B_FW_P9090"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>assign-fport.pp WS-B FW 9090
   </argument-string>
</command>

<command
 name="change-ref-fport_WS-A_WS-B_PC_FW_P9090"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-ref-fport.pp WS-A WS-B
   PC FW 9090</argument-string>
</command>

<command name="stop-service_WS-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-A
   </argument-string>
</command>

<command name="assign-fport_WS-A_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>unassign-fport.pp WS-A FW 8080
   </argument-string>
</command>

<command name="close-fport_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>close-fport.pp FW 8080
   </argument-string>
</command>
```

C Primitive ControlTier Commands of Cloud-Burst

```
<command name="open-fport_FW"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>open-fport.pp FW 8080
   </argument-string>
</command>

<command name="start-vm_VM-B_PRIV-CLOUD"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-vm.pp B PRIV-CLOUD
   </argument-string>
</command>

<command name="start-service_WS-B_VM-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-service.pp WS-B VM-B
   </argument-string>
</command>

<command name="change-ref_WS-A_WS-B_PC"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-ref.pp WS-A WS-B PC
   </argument-string>
</command>
```

```
<command name="stop-service_WS-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-A
   </argument-string>
</command>

<command name="stop-vm_VM-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-vm.pp VM-A
   </argument-string>
</command>

<command
 name="migrate_VM-A_PRIV-CLOUD_PUB-CLOUD"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>migrate.pp VM-A PRIV-CLOUD
   PUB-CLOUD</argument-string>
</command>

<command name="set-need-firewall_WS-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>set-need-firewall.pp
   WS-A</argument-string>
</command>

<command name="start-vm_VM-A_PUB-CLOUD"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-vm.pp VM-A PUB-CLOUD
   </argument-string>
</command>

<command name="assign-fport_WS-A_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>assign-fport.pp WS-A FW 8080
   </argument-string>
</command>

<command name="start-service_WS-A_VM-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-service.pp WS-A VM-A
   </argument-string>
</command>

<command
 name="change-ref-fport_WS-B_WS-A_PC_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-ref-fport.pp WS-A WS-A
   PC FW 8080</argument-string>
</command>

<command name="stop-service_WS-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-B
   </argument-string>
</command>

<command name="stop-vm_VM-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-vm.pp VM-B
   </argument-string>
</command>
```

Fine-grained access-control for the Puppet configuration language

Bart Vanbrabant, Joris Peeraer, Wouter Joosen
bart.vanbrabant@cs.kuleuven.be, jorispeeraer@gmail.com, wouter.joosen@cs.kuleuven.be
DistriNet, Dept. of Computer Science,
K.U.Leuven, Belgium

Abstract

System configuration tools automate the configuration and management of IT infrastructures. However these tools fail to provide decent authorisation on configuration input. In this paper we apply fine-grained authorisation of individual changes on a complex input language of an existing tool. We developed a prototype that extracts meaningful changes from the language used in the Puppet tool. These changes are authorised using XACML. We applied this approach successfully on realistic access control scenarios and provide design patterns for developing XACML policies.

1 Introduction

The management of large IT infrastructures needs to be automated to keep it manageable and reduce the amount of human errors [3, 11]. A system configuration tool is software that enables a system administrator to automate the configuration and management of large IT infrastructures. These tools address scalability, heterogeneity, and the consistency of relations between machines [2]. All system configuration tools have a similar reference architecture: each managed device runs an agent that manages the configuration of that device. The agent compares the current state of the device with the state described in a policy that is stored in a database or repository on a central server. This policy determines the configuration and the state of the entire IT infrastructure. Therefore if someone adds unauthorised changes to the central policy this person can control the entire IT infrastructure. Thus access control to this central policy is required.

System configuration tools can be divided into two categories based on how the input policy is organised: database or textual [5] based. The database based tools often use a graphical interface or command interface to manipulate their policies. Access control in these tools is enforced on records in that database. The other input

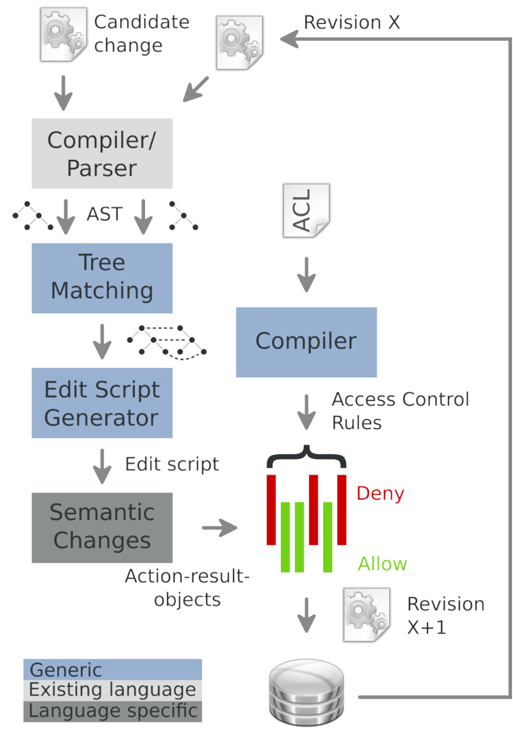


Figure 1: Overview of the solution presented in this paper.

type uses textual configuration files. The current state-of-practice of these text based system configuration tools uses path based access control to prevent unauthorised access to the textual configuration files but the name and the path of the file often do not have a relation with the contents of the file. To be able to use conventional path based access control, current tools rely on conventions to determine in which file what configuration statement may be included. For example network related configuration can only be defined in the `network.cf` configuration file. System management tools and path based

access control however cannot prevent a malicious user from adding network configuration statements to the file `motd.cf`.

In "Federated Access Control and Workflow Enforcement in Systems Configuration" [10] we proposed a method called ACHEL to enforce fine-grained access control based on the semantics of a change. In other words ACHEL calculates the operations the user wants to authorise using the changes in a textual file. We applied this approach to a minimal configuration language to prove its viability in a prototype. This prototype used a custom access control language based on regular expressions. Our method is language agnostic, except for the part to give meaning to each change. Figure 1 show the steps in the ACHEL method.

In this paper we apply the ACHEL method on a configuration language of an existing system configuration tool. The two research objectives of this paper are:

1. Can we extract the AST from the compiler of a system configuration tool and can we reuse this internal AST or do we need to transform it in order to obtain differences that are semantically meaningful?

2. How do we authorise changes? Once changes are known they have to be authorised. We propose an access control language and design patterns in this paper that provide the flexibility to express the different rules in a manageable and understandable fashion.

In this paper we add fine-grained access control based on meaningful changes to Puppet [1] and used XACML [8] to authorise changes. We implemented a prototype and integrated it with a version control system. Afterwards we evaluated this prototype by comparing it with traditional access control in two change scenarios. An important subset of the Puppet language is supported and we present design patterns to use the full expressiveness of the Puppet language including the unsupported language constructs with our access control mechanism.

In the remainder of this paper we first give some background on the ACHEL authorisation mechanism in Section 2. Then we look at related work in Section 3. Afterwards we discuss the methods used in the differencing process in Section 4. In this section we also discuss the problems we encountered by using the AST Puppet generates. The next step is the actual authorisation. Section 5 introduces the XACML framework and proposes a design pattern for writing policies. Finally in Section 6 we compare our method with other authorisation methods.

2 The ACHEL authorisation mechanism

The configuration model used by a system management tool is compiled from an input in the form of textual source code. This source input is stored on a filesystem or in a repository that uses version tracking. Access control and authorisation in the state of the art is based on operations performed on files and directories. In state of the art system management tools there is often no link between the file path and the parts of the configuration model represented in the file. Version control systems use diff-like algorithms [9] that operate on flat files to generate changes between two versions of a file. Diff algorithms detect changed lines and produce a list of insert and remove line operations. Applying access control on these operations does not make much sense. The operations are highly syntax dependant and there is only a weak link between the insert and remove operations and the configuration model.

In large infrastructures updates are never applied directly to the production infrastructure. Depending on the contents of the update or the person that produced the update, different authorisations can be required. For example:

1. all changes from junior administrators need to be reviewed and approved by a senior administrator

2. the scenario in Figure 2 where a change needs to be approved by a manager

3. all changes to the production infrastructure outside maintenance windows require approval by two managers

4. in a federated infrastructure changes to the backbone network need to be approved by the management of each of the administrative domains

Existing system management tools and access control solutions provide no support for these complex workflows.

Our method [10] transforms the updates on the configuration model by comparing the current and the new version of the input source. It compiles the two versions to an abstract syntax tree [7]. From the two versions an *edit script* is generated that transforms the old AST to the AST of the new version [4]. This process is represented in Figure 1. Because we are working on the AST, we know the semantics of changes made to the nodes in the abstract syntax tree. Therefore the edit script can be transformed to operations on entities that exist in the configuration model. Using our method, access control rules can be expressed in terms of operations on the entities in the configuration model. For example, instantiating a new resource, instead of adding a line to the input file that has a given syntax. These operations are the actions

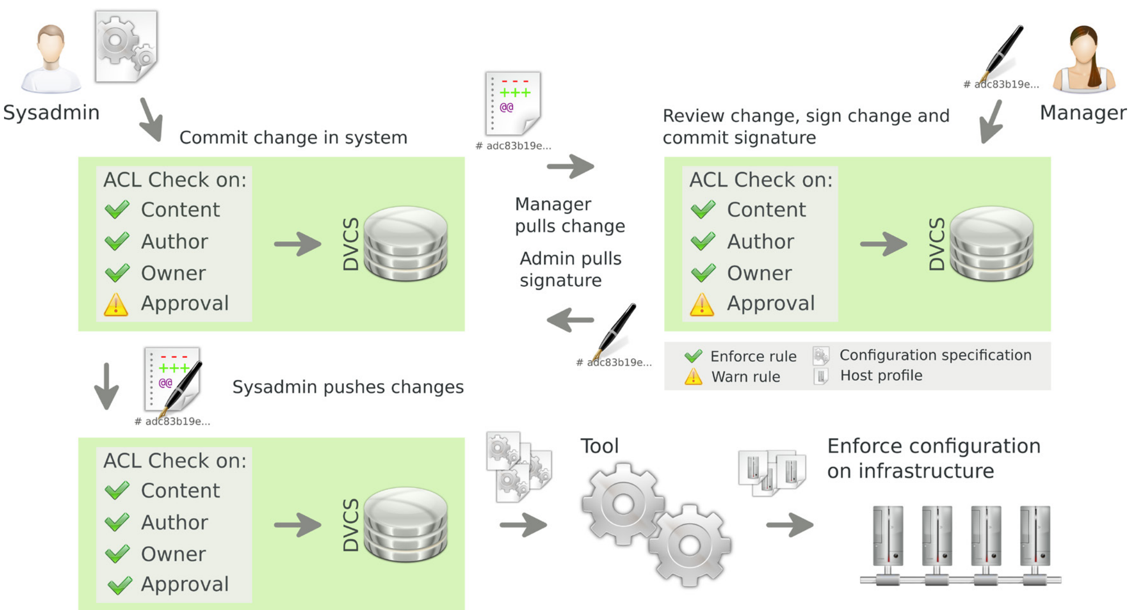


Figure 2: Updating the configuration model using access control.

that needs to be authorised. Additionally this method, opposed to other access control methods, derives the operations to authorise automatically and requests permission to apply them.

For audit purposes the configuration model is often stored in version controlled repositories. These repositories record each change to a configuration file and metadata such as the user that made the change and an optional log message. In ACHEL changes to this repository are digitally signed with the private key of the administrator. During generation of the edit script and the transformation of the edit script, the owner of each entity and the author of each change is tracked. The owner of an entity is the user that added or modified the entity. This ownership and author information is also exposed to the access control engine.

We enforce update workflows by using distributed version control repositories. Each system administrator that makes changes to the configuration model has their own repository. Distributed version control repositories assign a unique identifier to each change based on the contents of the change. To enforce update workflows, a change is authorised by the owner of a key by signing this unique identifier and including it as an update in the repository. Access control rules can require the authorisation of a third party before an update is allowed. Because each distributed version control repository can have its own set of access control rules, very flexible update workflows can be enforced.

Figure 2 represents a possible scenario supported by ACHEL [10]. A system administrator makes a change that is allowed in his repository but it requires approval by a manager to push the change into the repository for the production infrastructure. The sysadmin requests the manager to review his change. The manager reviews the change and approves it by signing the identifier of the change. The sysadmin can now push his change to the production repository together with the signature of the manager.

3 Related work

In "A survey of system configuration tools" [5] we evaluated several system configuration tools, including their support for access control and authorisation of changes. We identified two types of authorisation: either path based access control or access control based on "resources" in the configuration model. The tools that support external version repositories can reuse the path based access control of that repository or the access control models that the filesystem provides. Other tools allow fine grained access control on "resources" in a database using a hierarchy of resources. The system configuration tools that enforce authorisation on "resources" do this on resources in the configuration model that is used to generate and deploy configuration files and manage each system. The main disadvantage of this method is that authorisation cannot be performed on language constructions that are determined at runtime. For example the usage of a *Collect* instruction in Puppet.

"Authorisation and Delegation in the Machination Configuration System" [6] proposes a method of organ-

ising and delegating access to configuration information. The author integrated this method in the configuration management tool Machination. One of the key requirements of his method is the ability to authorise access to configuration aspects individually. He accomplishes this requirement by authorising the primitive operations which manipulate the configuration. The configuration representation used in Machination, is a form of XML with additional restrictions. These restrictions assure every configuration element is addressable by an XPath query. Upon this representation, a set of primitive operations is defined. These operations edit the configuration by adding, removing, changing and ordering the individual elements in the configuration input. Authorisation is then performed upon the individual operations needed to transform the configuration. By grouping multiple elements together such that they can be referred to by an XPath query, multiple configuration aspects can be authorised.

Both tools use the principle of authorisation on the individual elements. Where Machination starts from one version and uses the operation to obtain the new version, ACHEL derives the operations that need to be authorised from the two versions. This paper describes a method in which the differencing of ACHEL is used to find the changes made to a file.

4 Extracting changes

The ACHEL authorisation method starts from the configuration file that has been changed. The method consists of two phases. The first phase retrieves the AST of each file from the Puppet compiler. This AST should not contain any grammatical constructs anymore for the differencing algorithm to work. This is not the case for the AST Puppet produces, it still contains some syntactical leftovers. Therefore a transformation step is also included in this phase. The second phase compares the two trees and calculates an edit script that describes the operation to transform the first AST into the second AST. This edit script is transformed into meaningful changes expressed in terms of the language constructs in the Puppet language.

The Puppet language is an expressive language that also contains control flow and runtime evaluated expression such as the case statement or virtual and exported resources. Applying access control to changes that include these language constructs is very hard with our method because the effects of such statements are only known when the “configuration policy” of each managed device is calculated. In this prototype of the ACHEL authorisation method for Puppet we support a limited set of language constructs in the Puppet language on which we can apply authorisation. This set includes creating definitions, classes, creating resources including using arrays as identifier and relations. In Section 5.2 we will argue why this limited set can already be used to create powerful access control policies.

4.1 Generating the abstract syntax tree

The ACHEL authorisation mechanism requires access to the AST of each version of the Puppet manifests. In this section we describe how we extract the AST from Puppet. The AST from Puppet is not directly usable for our mechanism because it contains syntactical constructions. Therefore we need to normalise this AST.

We use the Puppet parser to create the AST of a Puppet input file. This provides us the AST that Puppet reasons upon. However this AST is not suited for generating an edit script. Although it is an *abstract* syntax tree, the tree still contain syntactical language constructs from the Puppet language. This is a problem because they do not have any meaning. This can even result in two different AST’s that have the same semantics. Consider Figure 3a and the corresponding AST in Figure 3b. Line 1 in Figure 3a describes the declaration of two users. The AST of the code fragment still contains the array which is nothing more than syntactic sugar to easily create two resources with the same attributes. For the differencing algorithm the only difference is the addition of one string to an array, instead of adding an entire resource. This change is not meaningful and cannot be described correctly in a policy.

The solution is to transform and normalise this tree to remove all syntactical structures from the abstract syntax tree. In the example from the previous paragraph we can remove the array as identifier for the users, and replicate the whole definition of the user for each element of this array. This transformation ensures that when a user gets added or removed, the differencing will detect a user being added or removed. The transformed AST is depicted in Figure 3c.

The solution in this example is very specific for the given problem and there is no generic solution to remove the syntax leftovers in the AST or even to detect them. Moreover, we do not have a list of problematic structures that are present in the Puppet language. The only solution to fully support a language is to test every language concept and check the resulting AST structure. In this implementation of our authorisation mechanism we explicitly chose to transform the AST instead of running a preprocessor over the input source. With this approach we reuse the existing lexer and parser of the Puppet tool. This makes the transformation step less syntax dependant. If a concept results in an ambiguous AST or one that contains syntactical constructs, the AST needs to be transformed or the compiler needs to be adapted.

```
1  user {["kwik","kwak"]:
2    gid => 123
3  }
```

(a) The Puppet manifest

```
1  class: ASTClass
2  - member: Resource
3    + type: Name => user
4    + title: ASTArray
5    | + child: String => kwik
6    | - child: String => kwak
7    - parameter: ResourceParam
8      + param: Name => gid
9      - value: String => 123
```

(b) The AST created by Puppet

```
1  class: ASTClass
2  + member: Resource
3  | + type: String => user
4  | + title: String => kwik
5  | - parameter: ResourceParam
6  |   + param: Name => gid
7  |   - value: String => 123
8  - member: Resource
9    + type: String => user
10   + title: String => kwak
11   - parameter: ResourceParam
12     + param: Name => gid
13     - value: Name => 123
```

(c) The normalised AST

Figure 3: Puppet configuration that defines multiple users using one resource definition.

The differencing stage compares the two normalised AST's to generate an edit script. In this prototype we use the same algorithm as our previous work [10]. This algorithm works as follows:

1. Match the leaves of the two trees using a similarity function.
2. Match the internal nodes using the information of already matched leaves: nodes with a lot of leaves in their subtrees in common are likely to match as well.
3. Correct wrongly coupled leaves using information of the matched internal nodes: parents of matching leaves should match as well.
4. Generate an edit script with the basic changes: add, modify and delete.
5. Correct changes: e.g. remove changes that cancel each other.

4.2 Generating meaningful changes

Authorisation is enforced based on operations derived from the meaning of a change and not on the operations the operations in an edit script, therefore the edit script is transformed in meaningful changes. For instance, the mode parameters of the of the `/etc/motd` changed from 0600 to 0644 instead of the 0600 node in the AST was removed and replaced by the 0644 node. These meaningful changes express changes as operations on the concepts that exist in the Puppet configuration language, instead of operations on a tree. This step is language dependant. The edit script expresses operations on the nodes in the AST. These nodes in the AST are linked to specific concepts in the Puppet language. In this step a transformation between the operations and the AST nodes and possible operations on language constructs is required. In our method this is a manually coded step.

5 Authorising changes

The second component in our solution is the authorisation of individual configuration changes. For this authorisation two elements are needed: a set of policies describing which changes are allowed or denied and a framework that executes the actual authorisation. XACML provides both features and is widely used authorisation standard in industry. Therefore we used it for implementing the authorisation step. In this section we will discuss the use of XACML to describe the access control policies.

5.1 The XACML standard

XACML is a international standard for access control and authorisation. The standard defines a language for policies and a language for authorisation requests. Both are XML based. The standard also describes the components and the architecture of an authorisation engine and allows an XACML authorisation engine to be extended.

XACML defines the components and the dataflow between them in the authorisation engine. The following components are required to handle an authorisation request:

- **Policy Enforcement Point (PEP)** This component receives the authorisation requests and creates a XACML request from it that is sent to the PDP.
- **Policy Decision Point (PDP)** The PDP loads all required policies and validates the request from the PEP against these policies. The results of these checks are combined and sent back to the PEP.
- **Policy Access Point (PAP)** The PAP makes the policies available to the PDP.

- **Policy Information Point (PIP)** The PIP provides the PDP with attributes related to subject, resource or environment. These attributes can be retrieved from several sources such as files or databases.

XACML is a generic solution for domain specific authorisation. The domain specific entities involved in the authorisation process can be mapped to subject, resource and action from the XACML standard. The subject submits a request to perform an action on a resource. Each of these entities can have multiple attributes. A policy decision is based on these attributes and additional attributes provided by the PIP.

The policy contains the rules that define what is allowed. Policies can be grouped in policy sets and each policy set can consist of policies and other policy sets. A policy is built from targets, rules, a rule combination algorithm and a number of obligations.

- The target of a policy defines when a policy needs to be used. This is expressed using a matching expression over the attributes of subject, action and resource.

- Rules have a target that defines when a rule is applicable, a condition and an effect that defines *Permit* or *Deny* based on the condition.

- The combining algorithm determines what the final result of a policy is if multiple rules returned an effect.

- The obligation is an action that needs to be executed when a policy is applicable. The PEP is responsible for executing these obligations.

The authorisation process works by exchanging request and response messages between the PDP and the outside world. The request message contains the subject, resource, action and environment and the associated attributes. If the content of the resource is XML it can be embedded in the request message. When the PDP has calculated the result of the authorisation a response message is sent back. This response message contains a result code and an optional message or information.

5.2 XACML policies for Puppet

Our authorisation method provides the configuration changes to the XACML engine that enforces authorisation. This section explains how a policy can reference Puppet language constructions in a configuration change. This section also provides a design pattern to encapsulate unsupported language constructions to enforce authorisation on them. The XACML standard describes an XML-based policy language. This language provides methods to access and compare attributes of the resource, subject and action involved in the authorisation-request. Complex functions can be used to process these attributes and to calculate the outcome of the policies. An example policy is shown in Figure 4.

XACML policies need a method for referring to the operations an update consists of and to the Puppet language constructs the operations act upon. Because XACML is based on XML and the configuration input is already available in the form of a abstract syntax tree. Additionally a resource can be embedded in a XACML authorisation request if the resource is represented in XML. Therefore we developed an XML serialisation of the Puppet manifests. We based this serialisation on the approach of Machination [6] to refer to constructs in the input using XPath. The AST is transformed into an XML tree that can be referenced uniquely by means of XPath expressions. Figures 5a, 5b and 5c show a Puppet statement and the two representations of the AST.

XPath queries can refer to the individual elements in the XML serialisation of the AST. When the node-id of a node is known, this attribute can be used to refer directly to this node using an XPath query like `//*[@id ="3"]`. When referring to the node in function of its attributes and location the following XPath query can be used: `//class[@name="apache"]/*[@type="package"]`

Puppet classes and definitions can be used to create abstractions on which access control can be enforced. These abstractions can encapsulate the language concepts our prototype currently does not support or that are very hard or impossible to support because of their dynamic nature. We used this design pattern in our evaluation the create access control policies. Superusers are allowed to make all changes, including the statements that are not supported. These superusers encapsulate these statements in definitions and classes that can be used by other users. This design pattern matches closely to the configuration module approach used by Puppet. These modules encapsulate the domain expert knowledge in easy to use interfaces and classes.

5.3 Using external information sources

XACML can use external sources for information through a PIP. In our prototype we extended the XACML engine to retrieve external information from directory services such as LDAP or active directory. These directories contained the roles of each user that can make changes. Storing this information in an external source and making it available in the XACML engine, makes it possible for the XACML policies to be more generic. Puppet also supports external sources for retrieving the classes that it should assign to hosts. One of such exter-

```
<Policy PolicyId="nodes:apache">
  <Target><Resources><Resource>
    <ResourceMatch MatchId="xacml:function:xpath-node-match">
      <AttributeValue DataType="xacml2:data-type:xpath-expression">
        //class[@name="apache"]
      </AttributeValue>
      <ResourceAttributeDesignator AttributeId="xacml:resource:resource-id"
        DataType="xacml2:data-type:xpath-expression" />
    </ResourceMatch>
  </Resource></Resources></Target>
  <Rule Effect="Permit" RuleId="nodes:apache:webadmin">
    <Target />
    <Condition>
      <Apply FunctionId="xacml:function:string-greater-than">
        <AttributeValue DataType="xs:string">xyz</AttributeValue>
        <Apply FunctionId="xacml:function:string-one-and-only">
          <SubjectAttributeDesignator DataType="xs:string"
            AttributeId="xacml:subject:subject-id"/>
        </Apply>
      </Apply>
    </Condition>
  </Rule>
</Policy>
```

Figure 4: A sample XACML policy file for configuration changes.

```
# Apache-class
class apache inherits webserver {
  package {"apache": ensure => installed }
}
```

(a) Sample Puppet configuration

```
Root
+ hostclasses: ResourceType ()
| - class: ASTClass (name:apache)
|   + parent: Name () => webserver
|   - member: Resource (title:apache,type:package)
|     + parameter: ResourceParam (param:ensure)
|     - value: Name () => installed
- nodes: ResourceType ()
```

(b) The abstract syntax tree

```
<Root id='1' nodetype='ASTRoot' xmlns='pupa'>
  <hostclasses id='2' nodetype='ResourceType' >
    <class id='3' nodetype='ASTClass' name='apache'>
      <parent id='4' nodetype='Name' >webserver</parent>
      <member id='5' nodetype='Resource' title='apache' type='package'>
        <parameter id='7' nodetype='ResourceParam' param='ensure'>
          <value id='8' nodetype='Name' >installed</value>
        </parameter>
      </member>
    </class>
  </hostclasses>
  <nodes id='9' nodetype='ResourceType' >
  </nodes>
</Root>
```

(c) The XML representation of the AST

Figure 5: Puppet configuration file and the resulting AST and its XML representation

nal sources is an LDAP directory. This information can also be exposed in the XACML engine through a PIP.

6 Evaluation

We evaluated our prototype based on two access control scenarios. These scenarios each describe a policy that has to be enforced. In the evaluation we construct a policy-file that tries to accomplish this task and explain the reasons behind its structure. We compare the results of our policy with a policy based on path based access control available in version control systems. The goal of these evaluations is to show the possibilities and limitations of our tool.

For this evaluation we integrated our prototype into a version control system (VCS). The VCS is used as storage for the configuration files and also acts as the authorising agent. Additionally a Policy Information Point (PIP) provides additional attributes to the XACML engine. The PIP manages information and is contacted during the authorisation process when specific attributes are needed. The PIP in our implementation connects to an LDAP server and provides attributes belonging to the administrator issuing the change.

In the evaluation, two scenarios are investigated:

1. Only system administrators that are members of the webadmin group can configure a machine as an apache webserver.
2. Only system administrators that are members of the webadmin group can create a virtual host on an apache webserver. Moreover, the documentroot of the virtual host can only exist inside the homedirectory of the user issuing the change.

6.1 Scenario 1: configure a machine as webserver

In the first scenario we have a simple Puppet configuration that configures nodes as a webserver. In the Puppet module path an apache module is added with a class named `apache`. The site.pp file contains a list of nodes that each have a list of include statements that add functionality to that server. Figure 6 shows the initial site.pp file for this scenario. The change in this scenario configures the spare server *san-jose* as webserver by including the `apache` class from the apache module. The security policy says that all administrators can add *roles* to servers by including classes, but only users from the group *webadmin* may configure a server as webserver.

A SVN repository can be used to limit access to the file in the repository. Figure 7 shows a file with access control rules for this repository. The puppet user has read-only access to the site.pp file in the site directory and has access to the files in the apache module. Only webadmin users can edit the files in the apache module, but this does not prevent them from including for example a statement that installs a dns server in that module. The site.pp file can be edited by all administrators. The access control mechanism from SVN cannot prevent non-webadmin users to create new webservers either.

Figure 8 shows the XACML policy for the ACHEL authorisation mechanism. This policy allows users from the webadmin group to include apache classes in the configuration of a node. This XACML policy provides a very fine-grained access control to the statements in the Puppet manifests.

```
1  node baltimore {
2    # include the apache module
3    include apache
4  }
5
6  node san-jose {
7    # spare machine to configure as
         webserver
8  }
```

Figure 6: The site.pp file for Puppet for the first scenario. One server is already configured as webserver. The other server is kept as spare server.

```
1  [/modules/apache]
2  puppet = r
3  @webadmin = rw
4
5  [/site]
6  puppet = r
7  @admins = rw
```

Figure 7: An authorisation file for SVN to restrict access to the Puppet manifests in the repository.

6.2 Scenario 2: add virtual hosts

The second scenario uses the same apache webserver setup and allows users from the webuser group to add virtual hosts to the apache configuration. Webusers can only add virtual hosts to the configuration from which the document root is located in their own home directory. The document root parameter controls the directory that contains the files that a webserver should serve to visitors of a particular domain. The home directory path of users is always built as follows: `/home/` and their username concatenated to that. The Puppet manifest in

```
<Policy PolicyId="nodes:apache">
  <Target><Resources><Resource>
    <ResourceMatch MatchId="xacml:function:xpath-node-match">
      <AttributeValue DataType="xacml2:data-type:xpath-expression">
        //p:node/p:include[@class="apache"]
      </AttributeValue>
      <ResourceAttributeDesignator
        AttributeId="xacml:resource:resource-id"
        DataType="xacml2:data-type:xpath-expression" />
      </ResourceMatch>
  </Resource></Resources></Target>
  <Rule Effect="Permit" RuleId="nodes:apache:webadmin">
    <Target><Subjects><Subject>
      <SubjectMatch MatchId="xacml:function:anyURI-equal">
        <AttributeValue DataType="xs:anyURI">webadmin</AttributeValue>
        <SubjectAttributeDesignator AttributeId="xacml2:subject:role" DataType="xs:anyURI
            " />
      </SubjectMatch>
    </Subject></Subjects></Target>
  </Rule>
</Policy>
```

Figure 8: The XACML policy to allow users from the webadmin group to add the apache class to a node.

Figure 9 shows a manifest from the apache module that configures the virtual hosts in the system.

The access control configuration for a SVN repository for this scenario is similar to the previous scenario. Figure 11 shows an updated authorisation file for the SVN repository that contains the Puppet manifests for this scenario. This file limits access to the vhosts.pp file to users from the webuser group. It cannot prevent users from adding virtual hosts that have a document root in the home directory of another user. It also does not prevent users from adding virtual host resources to other manifest files.

The XACML policy in Figure 10 enforces access control based on the contents of the change and not based on the file location. It enforces access control on all occurrences of virtual host resources in any Puppet manifest file that is included in the repository. The policy builds the home directory of the user by concatenating */home/* with the username. The value of the documentroot parameters of the virtual host resource should always start with the home directory string the policy created.

This policy can be circumvented by using a path that starts with the users home directory but then uses `..` to traverse back to the `/home` directory. For example, `/home/lisa/../foo/www`. This is not a flaw of the approach but a limitation of the expressiveness of the XACML functions. To close this policy a function that normalises the path first before it does the compare is required. In the conclusion we will discuss how to counter this attack.

```
class apache {
  apache::vhost {"www.example.com":
    docroot => "/home/lisa/www"
  }

  apache::vhost {"photo.example.com":
    docroot => "/home/lisa/photo"
  }
}
```

Figure 9: The vhosts.pp file for Puppet for the second scenario that is located in the /vhosts directory.

```
[/modules/apache]
puppet = r
@webadmin = rw

[/vhosts]
puppet = r
@webuser = rw

[/site]
puppet = r
@admins = rw
```

Figure 11: An update authorisation file for SVN for the second scenario.

6.3 Conclusion

The conclusions from this evaluation are twofold. First, there is a strong need for content-aware authorisation.

```
1  <Policy PolicyId="apache:webuser">
2    <Target><Subjects><Subject>
3      <SubjectMatch MatchId="xacml:function:anyURI-equal">
4        <AttributeValue DataType="xs:anyURI">webuser</AttributeValue>
5        <SubjectAttributeDesignator
6          AttributeId="xacml2:subject:role" DataType="xs:anyURI" />
7      </SubjectMatch>
8    </Subject></Subjects></Target>
9    <Rule Effect="Permit" RuleId="apache:webuser:vhost">
10     <Description>Add or remove a vhost</Description>
11     <Target><Resources><Resource>
12       <ResourceMatch MatchId="xacml:function:xpath-node-equal">
13         <AttributeValue DataType="xacml2:data-type:xpath-expression">
14           //pup:*[@type="apache::vhost"]
15         </AttributeValue>
16         <ResourceAttributeDesignator AttributeId="xacml:resource:resource-id"
17           DataType="xacml2:data-type:xpath-expression" />
18       </ResourceMatch>
19     </Resource></Resources></Target>
20   </Rule>
21   <Rule Effect="Permit" RuleId="apache:webuser:vhost-docroot">
22     <Target><Resources><Resource>
23       <ResourceMatch MatchId="xacml:function:xpath-node-match">
24         <AttributeValue DataType="xacml2:data-type:xpath-expression">
25           //p:*[@type="apache::vhost"]/p:parameter[@param="docroot"]
26         </AttributeValue>
27         <ResourceAttributeDesignator AttirbuteId="xacml:resource:resource-id"
28             DataType="xacml2:data-type:xpath-expression" />
29       </ResourceMatch>
30     </Resource></Resources></Target>
31     <Condition><Apply FunctionId="thesis:function:string-starts-with">
32       <Apply FunctionId="xacml:function:string-one-and-only">
33         <AttributeSelector DataType="xs:string"
34             RequestContextPath="//p:param[@param='docroot']/p:value/text()" />
35       </Apply>
36       <Apply FunctionId="xacml2:function:string-concatenate">
37         <AttributeValue DataType="xs:string">/home/</AttributeValue>
38         <Apply FunctionId="xacml:function:string-one-and-only">
39           <SubjectAttributeDesignator DataType="xs:string"
40             AttributeId="xacml:subject:subject-id" />
41         </Apply>
42       </Apply>
43     </Apply></Condition>
44   </Rule>
45 </Policy>
```

Figure 10: The XACML policy that only allows users from the webuser group to add vhosts with a document root in their homedirectory.

Our prototype is able to provide this by analysing the changes made to a configuration file and deriving the actions from it that need to be authorised. The prototype is flexible enough to do this in a very fine-grained manner. Using the ACHEL method changes to Puppet manifests are authorised at the level of instantiating resources and authorising them based on the parameters and the scope they are declared in.

Second, our prototype is not yet fully finished. It is possible to circumvent the authorisation using simple attacks. This is not a limitation of our approach but of the XACML language expressiveness which results in a policy that is not fully closed. A possible solution for this type of attacks is extending the standard XACML function-space to include domain-specific helper functions to create a fully closed access control policies. In our prototype we used the enterprise-java-xacml [12] XACML engine. Adding functions to this engine is easy as adding an annotation to a Java class and the method that implements the XACML function.

7 Future work

We added basic support for our authorisation mechanism to Puppet. Future work in this direction should focus on extending support for the Puppet language and on integrating update workflows in the authorisation phase.

Extending Puppet support Currently Puppet support is limited to a subset of the Puppet language. This subset provides a usable implementation, especially using the design patterns we described. Our prototype can be extended to support additional language constructs such as calling functions or exporting resources.

Integrating workflow In our original paper [10] we also integrated workflow support. This support is orthogonal to extracting meaningful changes from changes in Puppet manifests. Therefore we did not implement this in this prototype. This workflow support is based on digital signatures on revisions in the version repository. This information can be made available to the XACML engine through a PIP. This should be sufficient to add the workflow support to this prototype.

Ownership information Our ACHEL method can also track ownership information based on the changes made to the configuration files. With this ownership information a policy could also include that person A cannot change any parameters or resources that are owned by person B. To derive ownership information we need to start from the first revision and determine for each change what the impact is on the ownership of a statement. For Puppet one of these questions is who is the owner of a resource? Is this the user that gave the name to the resource? If parameters are changed, how does the ownership of the resource and the parameter change?

8 Conclusion

In this paper we developed a prototype that authorises changes to the Puppet input language based on their meaning. It derives the operations that need to be authorised from the changes to the input. This prototype extends our previous work by applying it to a real system configuration tool with a complex input language. These changes are authorised using XACML which is a widely adopted industry standard for access control and authorisation, instead of using regular expressions.

In our previous work we applied our approach to a simple configuration language. The results from this work show that although the configuration language is more complex, it is possible to extract the individual meaningful configuration changes. The complete language is not yet fully supported, and the prototype needs more work to analyze the difficult language constructs before it can be used in production. Our claim from our previous paper that the method is language agnostic holds, on the condition that the method can start from a clean AST. The usage of the XACML standard for authorisation provides a lot of flexibility for writing policies, as well as extensibility and integration of other information sources. This prototype does not include the workflow enforcement of our Achel [10] prototype but can be easily added to the XACML engine by including a PIP that provides the signature information required to enforce workflows.

This implementation uses XACML as authorisation language. In our first prototype we used a regular expression based language we developed. XACML provides an authorisation engine that is the de facto industry standard in contrast to our own authorisation language. The usability of our regular expression based language was also very poor. Unfortunately it appears to be hard to write policies in XACML as well. Luckily tooling support exists for writing XACML policies to improve usability.

To conclude the main contributions of this paper are: First of all the identification of the difficulties and possibilities to extract meaningful changes from a complex configuration language such as Puppet. Second the development of a set of rules that describe how to write a policy and refer to the configuration elements in these policies.

9 Acknowledgements

This research is partially funded by the Agency for Innovation by Science and Technology in Flanders (IWT), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

[1] Puppet Website. `http://www.puppetlabs.com`, 2010.

[2] ANDERSON, P. *Short Topics in System Administration 14: System Configuration*. Berkeley, CA, 2006.

[3] BARRETT, R., MAGLIO, P. P., KANDOGAN, E., AND BAILEY, J. Usable autonomic computing systems: The system administrators' perspective. *Advanced Engineering Informatics 19*, 3 (2005), 213 – 221. Autonomic Computing.

[4] CHAWATHE, S. S., AND GARCIA-MOLINA, H. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data - SIGMOD 97 SIGMOD 97* (New York, NY, USA, 1997), ACM, pp. 26–37.

[5] DELAET, T., JOOSEN, W., AND VANBRABANT, B. A survey of system configuration tools. In *Proceedings of the 24th Large Installations Systems Administration (LISA) conference* (San Jose, CA, USA, 11/2010 2010), Usenix Association, Usenix Association.

[6] HIGGS, C. Authorisation and Delegation in the Machination Configuration System. In *Proceedings of the 22nd Large Installation System Administration (LISA) Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 191–199.

[7] MCCARTHY, J. Towards a mathematical science of computation. *Information Processing 62* (1962), 21–28.

[8] MOSES, T. *eXtensible Access Control Markup Language (XACML) Version 2.0*, februari 2005. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

[9] MYERS, E. W. An O(ND) difference algorithm and its variations. *Algorithmica 1*, 1 (1986), 251–266.

[10] VANBRABANT, B., DELAET, T., AND JOOSEN, W. Federated access control and workflow enforcement in systems configuration. In *Proceedings of the 23rd Large Installations Systems Administration (LISA) conference* (Baltimore, MD, USA, 11/2009 2009), Usenix Association, Usenix Association, p. 129–143.

[11] VELASQUEZ, N. F., AND WEISBAND, S. P. Work practices of system administrators: implications for tool design. In *CHiMiT '08: Proceedings of the 2nd ACM Symposium on Computer Human Interaction for Management of Information Technology* (New York, NY, USA, 2008), ACM, ACM, pp. 1–10.

[12] WANG, Z. Enterprise Java XACML Implementation. `http://code.google.com/p/enterprise-java-xacml/`, december 2010.

tiqr: a novel take on two-factor authentication

Roland M. van Rijswijk – SURFnet bv, Utrecht, The Netherlands
Joost van Dijk – SURFnet bv, Utrecht, The Netherlands

ABSTRACT

Authentication is of paramount importance for all modern networked applications. The username/password paradigm is ubiquitous. This paradigm suffices for many applications that require a relatively low level of assurance about the identity of the end user, but it quickly breaks down when a stronger assertion of the user's identity is required. Traditionally, this is where two- or multi-factor authentication comes in, providing a higher level of assurance. There is a multitude of two-factor authentication solutions available, but we feel that many solutions do not meet the needs of our community. They are invariably expensive, difficult to roll out in heterogeneous user groups (like student populations), often closed source and closed technology and have usability problems that make them hard to use. In this paper we will give an overview of the two-factor authentication landscape and address the issues of closed versus open solutions. We will introduce a novel open standards-based authentication technology that we have developed and released in open source. We will then provide a classification of two-factor authentication technologies, and we will finish with an overview of future work.

1 INTRODUCTION

1.1 AUTHENTICATION

Authentication is something we all do every day. And whether it is to log in to our e-mail account, to access our Facebook page or to tweet about that cool new album we've just bought, the use of username/password is by far the dominant paradigm.

There are – of course – applications that require a higher level of assurance such as electronic banking. The traditional approach for achieving this higher level of assurance is to use multi-factor authentication (also referred to as *strong authentication*). There is a multitude of multi-factor authentication solutions on the market. Traditionally, this market has a strong tendency towards closed solutions with strong vendor lock-in. This invariably leads to a high cost per user, hampering the wide-scale rollout of multi-factor authentication technologies. Another common limitation of current multi-factor authentication technologies is the fact that they are often single-purpose solutions (e.g. they can only be used for one bank). Furthermore, there are serious usability issues with many multi-factor solutions that make it difficult to enforce their use in most communities.

Exactly what an acceptable level of assurance is should not only be decided by a service provider; users may also have an opinion on this. Almost all solutions currently on the market give very little control to the end user.

1.2 RECENT INDUSTRY DEVELOPMENTS

In recent years there have been some promising developments in the industry. In 2004, the Initiative for Open Authentication (OATH, [1]) was formed. The intention of this initiative is to create an industry-wide reference architecture for strong authentication. The OATH initiative has been very successful in creating industry standards for two-factor authentication that have been embraced by the Internet community in the form of IETF standards ([2], [3], [4]). A number of companies and open source initiatives have adopted these standards in products and services (see also section 3).

Other developments have underlined the need to adopt open standards in the security and authentication/identity management industry. Time and again closed solutions and algorithms have been shown to be vulnerable to attack because of the lack of peer review (poignant examples include the MIFARE system and the GSM A5/1 cryptographic algorithm [5]).

Finally, large players on the Internet have recently introduced two-factor authentication for some of their services ([6], [7]). This is the first time two-factor authentication is deployed on a (potentially) large scale for applications outside the financial industry or enterprise domain.

1.3 OVERVIEW OF THIS PAPER

In this paper we aim to give an overview of the current two-factor authentication landscape in section 2. In section 3, we will further clarify some of the issues we believe exist in current two-factor authentication market offerings. Section 4 proposes a way to classify authentication solutions and con-

tains a classification of the solutions discussed in section 2. In section 5, we will introduce the innovative two-factor authentication solution we have developed which is based on open standards and open technology. Section 6 revisits the classification proposed in section 4 and adds a classification for the solution we introduced in section 5. Finally, in section 7 we will draw conclusions and provide suggestions for future work.

2 THE TWO-FACTOR LANDSCAPE

2.1 INTRODUCTION

In this section we aim to give an overview of the two-factor landscape. Before we do that, we will first give a definition of what we think constitutes two-factor authentication.

We then describe the solutions currently on offer, which we divide into two categories:

- Traditional solutions – these rely on single purpose (i.e. only used for identification) hardware devices or on a unique quality of the user (i.e. a biometric)
- Hybrid solutions – these rely on non-single purpose devices owned by the user, possibly in combination with software running on these devices

2.2 DEFINITION OF TWO-FACTOR AUTHENTICATION

In this paper, we define two-factor authentication as a means of authentication relying on the user demonstrating at least 2 separate factors from the following list:

- Something the user *knows* (e.g. a PIN code or a password)
- Something the user *has* (e.g. a hardware token)
- Something the user *is* (e.g. a biometric, such as a fingerprint)

Solutions that we place in the "hybrid" category rely on something the user has but where there is a chance of this factor being duplicated as could – for instance – be the case with a soft token running as an application on a smartphone. Some people in the blogosphere have coined the term "1.5 factor authentication" for this category (e.g. [40])

In this paper we will refer to a solution as two-factor authentication whenever the device on which the user is authenticating is physically separate from whatever constitutes the second factor (e.g. a soft token on a phone is only a second factor if it is used for authenticating a session on a separate device such as the user's computer).

2.3 TRADITIONAL SOLUTIONS

2.3.1 OTP TOKENS

One-Time Password or OTP tokens are devices that generate single-use passwords (often composed of strings of up to 10 digits). There are two variants: time-based tokens – these generate a new password at regular intervals (e.g. every 30 seconds) and event-based tokens – these generate a new password after a user intervention (e.g. pushing a button on the device).

The second factor most often combined with these devices is either a password that is entered on the user's computer or a PIN that is entered on the token device itself.

OTP tokens rely on symmetric cryptography for their operation; they contain some secret that is securely stored in the device, which can never leave it. The same secret is also known on the server that validates the user's credentials when they log in.

Examples of OTP tokens include: VASCO Digipass [10], RSA SecurID [11] and Feitian OTP Tokens [12].

Figure 1 - example of an OTP token (RSA SecurID)

2.3.2 CHALLENGE/RESPONSE TOKENS

Challenge/response tokens are similar to OTP tokens in that they also rely on symmetric cryptography for their operation. Some OTP tokens also have challenge/response capabilities.

Whereas OTP often suffices for simple authentication, challenge/response tokens are mainly used for transaction authentication such as, for instance, approving a money transfer. This is achieved by having the user enter one or more sequences of digits on the token (the challenge) and using these as input for a cryptographic algorithm to produce another sequence of digits (the response) that the user then returns to the party requesting authentication.

Challenge/response tokens are usually protected using a PIN code as the second factor.

Examples of challenge/response tokens include VASCO DigiPass [13], SafeNet SafeWord GOLD [14] and Feitian Challenge/Response tokens [12].

Figure 2 - example of a challenge/response token (SafeWord GOLD)

2.3.3 PKI TOKENS

In contrast to the previous two solutions, PKI tokens rely on public key cryptography.

Under the hood, almost all PKI tokens rely on smart card ICs with a cryptographic co-processor capable of performing public key operations and – in most cases – key generation. They come in a variety of form factors, the two most common being the smart card and the USB dongle.

Authentication with PKI tokens usually relies on some form of challenge/response algorithm. The aim of these algorithms is to prove that the user is in possession of the private key belonging to a public key that is usually stored in an X.509 certificate (for more details, see [5] sections 3.2 and 4).

Contrary to the previous two solutions, PKI tokens usually interface with the end user system. They rely on software running on that system to integrate with, for example, the browser and mail client. There is an exception to this rule: Mobile PKI (see [8]). In Mobile PKI, the user's SIM card is used as a PKI token; interfacing with the token takes place using special SMS text messages.

PKI tokens have a broader applicability than just authentication. They can also be used to create advanced – and in some jurisdictions legally binding – digital signatures (for more information, see [5] section 4.9).

2.4 HYBRID SOLUTIONS

2.4.1 SMS OTP

For many years now, the fact that almost everyone has a mobile phone is being used as a means for two-factor authentication. Many users will be familiar with SMS One-Time Passwords.

SMS OTP relies on an authentication server sending one-time passwords by SMS text message to the user. The user's mobile phone is thus leveraged as an authentication factor. The other factor is commonly username/password (thus the user first logs in using username/password and then provides additional proof of his or her identity using SMS OTP).

There is some discussion about whether SMS OTP constitutes real two-factor authentication ([15], [16]). Especially the fact that it is hard to protect the user against (temporary) stealing of their phone is a concern (putting a PIN lock almost never provides protection since SMS's are displayed even if the handset is locked).

There are many vendors of SMS OTP services; a Google search for "SMS OTP" will produce a long list.

2.4.2 OTP APPS

Another more recent development is the appearance of One-Time Password Apps. These run on modern handsets (smart phones) and usually mimic the behaviour of OTP tokens (see section 2.3.1). The difference between these Apps and 'real' OTP tokens is that the secret is stored and processed in software on the handset. This makes them somewhat more vulnerable to attacks.

Most OTP token vendors now also have App versions of their OTP tokens that interface with the same backend server systems that are also used for their hardware tokens.

3 ISSUES IN TWO-FACTOR AUTHENTICATION

3.1 INTRODUCTION

We feel that there are several issues surrounding two-factor authentication that are hampering rollout on a larger scale; most solutions are closed, they often use single-purpose tokens, are not easy to use, may have prohibitive costs associated with them and almost always lack user control. We will address these issues in more detail in the remainder of this section.

3.2 CLOSED SOLUTIONS

The most important issue with most current solutions is that they are closed ecosystems. For example, the majority of OTP tokens is based on proprietary algorithms and can only be integrated into applications by using servers or server-side components supplied by the token vendors.

Ironically, for PKI tokens it is even worse. They always require integration software on the client system in the form of cryptographic middleware (although they normally do not require server-side integration, since they are based on built-in X.509 client authentication). If the tokens are smart cards, they require smart card readers (which are not commonly installed in systems apart from some enterprise-market laptops). And both smart card readers as well as USB tokens may require specific drivers before they will work although that is less common nowadays with most of them supporting the CCID [17] standard.

Because most solutions require proprietary software, they are not easily integrated on all platforms (i.e. they will only work on vendor-supported platforms).

For OTP tokens, the advent of the OATH initiative brings hope since both the algorithms in the tokens as well as the way that the token secrets are distributed are now specified in open standards. This makes it possible to develop the server-side integration software independent of the token vendor and allows these components to support tokens from many vendors. There is already quite a bit of uptake among token vendors.

In contrast, for PKI tokens, the situation is different. Although there is an open source initiative [21], this project has not really seen a wide use or deployment and indeed most PKI token middleware is still proprietary and closed. On a positive note, PKI middleware at least adheres to the open PKCS #11 standard [22].

One final thing to mention is Mobile PKI. From an integration perspective it is fully open, because it is based on an open standard web service interface called MSSP [23], [24], [25], [26]. The downside is that a special application needs to be installed on the user's SIM card. The mobile operator owns the SIM card and access to it is strictly guarded. This means that in order to be able to deploy Mobile PKI co-operation of the mobile operator is required, which has been proven to be difficult on many occasions.

3.3 SINGLE PURPOSE TOKENS

Almost all OTP tokens are single-purpose tokens by nature because they rely on a shared secret. The tokens themselves can only contain one secret, which means that they can only be paired with one server. Unless the server is used as an authentication service for multiple applications (which is very rarely the case), the tokens can thus only be used for a single purpose (e.g. to log in to online banking for a single bank). This is very inconvenient for users, and indeed many users will know the hassle of having more than one token because they are customers at more than one bank.

In principle, PKI tokens should be more flexible because they often support storage of more than one X.509 certificate together with the associated key-pair. Unfortunately, the issuance process of PKI tokens is usually such that users have no control over the content of their token and can very rarely add credentials for additional identities. Thus, PKI tokens can only be used for multiple purposes if they contain an identity issued by a Certificate Authority that is supported by the party to which the user is authenticating.

In theory, mobile App-based solutions can more easily support multi-purpose deployments, in practice this does not happen very often yet.

3.4 (LACK OF) EASE OF USE

Many users will have experienced how difficult it can be to use OTP tokens. Most of them require typing in complicated codes. The challenge/response variety is even more complicated where users regularly have to type multiple codes on the token and then they have to copy the result from the token by typing it on the site they are authenticating to.

SMS OTP is no better. In fact, it is even more complicated in our opinion as the one-time passwords used often consist of both capitals and lower case letters as well as digits and punctuation marks.

PKI tokens fare a little better. As long as the software integration with the user's browser is properly installed, the user experience is usually quite smooth.

A common issue shared by all tokens except mobile phone-based ones is that they are all too easy to forget or lose.

3.5 COST

Both OTP and PKI tokens can be quite costly, both in initial investment as well as yearly licence fees. It is not uncommon to pay tens of US dollars per user per year. SMS OTP becomes gradually more costly the more it is used.

The only exception to this rule is a new class of OTP tokens that are emerging, based on open standards developed by the OATH initiative. Because they work with open source software, the only substantial cost is the initial investment. Yubikey tokens [27], for example, can be purchased for less than USD $30 and the price goes down for larger volume purchases.

3.6 (LACK OF) USER CONTROL

Users seldom initiate deployment of two-factor authentication solutions. They are usually deployed by corporate IT departments or banks. The organisations deploying these tokens strictly control what they can or cannot be used for, severely limiting users.

It is very hard for users to acquire personal two-factor tokens and deploy them in a useful way because very few services provide the means to self-enrol identities. A notable exception to this is Google Authenticator [28].

4 CLASSIFICATION OF AUTHENTICATION SOLUTIONS

4.1 INTRODUCTION

In this section we will introduce six different ways to classify authentication solutions in order to judge their suitability. We will use this classification at the end of this section to classify the two-factor authentication solutions discussed earlier.

4.2 HARDWARE INDEPENDENCE

The first way to classify authentication solutions is by their dependence (or lack thereof) on specific or specialised hardware for their operation.

We feel that hardware independence enhances the usability of a solution, because the more independent a solution is from specific hardware, the fewer devices a user has to carry around.

From a security perspective, however, using special purpose-made hardware has distinct advantages. Devices can be tailored for one goal, which is to protect the secrets associated with a user's credentials.

In this paper, we will focus mainly on the enhanced usability that comes with hardware independence; we will factor in the security advantages that special hardware can offer when we judge the security of a solution. We rank solutions that offer stronger hardware independence more favourably than solutions that require specific hardware to operate.

4.3 SOFTWARE INDEPENDENCE

Just like hardware independence, software independence is mainly a usability enhancing aspect. In some cases, dependence on specific hardware goes hand in hand with dependence on specific software. For example, smart cards cannot operate without the accompanying security middleware that users will have to install on their computer.

Some solutions only depend on specific software on the server side and do not require the user to install software (for example OTP tokens).

We will judge solutions on the amount of effort required to install software by both end users as well as by the system administrators of the server side. We will also factor in the availability of integration in off-the-shelf products as this can significantly reduce the effort required to install the required software.

4.4 SECURITY

Security is – of course – one of the most important factors when judging authentication solutions.

There are several aspects that influence the security of a solution:

- Is the solution a multi-factor solution? If so, is it a true multi-factor solution (see §2.3) or a hybrid solution (see §2.4)?
- Does the solution rely on purpose-built hardware that has provisions for e.g. tamper resistance?
- Are there well-known attacks that (severely) impact the security?
- If the solution relies on cryptography, does it rely on sufficiently strong as well as open cryptography?
- Has the security of the solution been verified by reputable independent security auditors?

4.5 COST

Cost is an important factor, especially for large-scale deployments. It can be considered from a number of different angles:

- The one-time setup cost (e.g. in software and hardware purchases) and recurring cost of the actual solution (e.g. yearly licence fees).
- The cost for troubleshooting for users who have misplaced their credentials or forgotten their password or PIN.
- The cost of integrating the solution into existing IT infrastructure (what skill level is required and how much time do system administrators or system integrators spend setting up the solution).

4.6 OPEN STANDARDS COMPLIANCE

Open standards form the backbone of the Internet. Vendors implement these standards that are available free-of-charge or for a reasonable fee to guarantee interoperability with systems from other vendors.

There is a whole host of open standards in the authentication arena that make it easier to integrate solutions into existing IT infrastructure. They also offer a certain level of vendor independence as one solution can be more easily exchanged for another. Of course, this also depends on the level to which open standards have been integrated. For example: OTP tokens that fully support the open standards of the Open Authentication Initiative can easily be integrated with server-side software from a range of vendors that support these standards. On the other hand, PKI tokens that rely on PKCS #11 middleware are less easily replaced by another solution as they will require specific middleware supplied by the token vendor.

For a long time supporting open standards was not common practice, especially among OTP token vendors. Fortunately, this is now changing for the better with the advent of consortia like the Open Authentication Initiative.

4.7 EASE-OF-USE

A final factor that can go a long way in determining the success of a solution is ease-of-use. At first glance, solutions that are already familiar to a user – such as username/password – may seem easy-to-use. But when all the kludges that have been added to enhance the security such as complex password policies and requirements to change passwords on a regular basis are considered, it is easy to see that such solutions may not be as easy-to-use as initially assumed.

Other things that need to be factored in when considering the ease-of-use of a solution are:

- Does the solution require the user to carry around additional devices (that he/she otherwise would not need to operate their computer)?
- Does the user have to re-type complicated codes (such as may be the case for OTP tokens)?
- Has care been taken to design the user experience such that the solution can be used intuitively by the user rather than requiring them to learn how to operate the solution from e.g. a manual?

4.8 CLASSIFICATION

Table 1 below shows the scores we have assigned to each solution described in section 2 for each of the 6 different classification categories described earlier; we used a five point scoring system ranging from ++ (indicating that a solution is (one of) the best in class for the given classification category) to -- (indicating that a solution has very unfavourable characteristics compared to other solutions in the given classification category). Any scoring system is, of course, subjective; we endeavour to justify the scores in Table 1 in section 4.9.

	Hardware indep.	Software indep.	Security	Cost	Open Standards	Ease-of-use
Usern./pwd	++	++	--	++	=	+/-
OTP token	-	-	++	--	-/=	+
C/R token	-	-	++	--	-/=	+
PKI token	--	--	++	--	=	+
Mobile PKI	+	+	++	?	+	++
SMS OTP	+	=	-	-	--	-
OTP Apps	+	+/=	+	+/=	+/=	=

Table 1 - Classification of authentication solutions

4.9 JUSTIFICATION

We would like to highlight certain points of the classification we made in the previous section. Given the endless stream of news articles about username/password getting compromised we feel that – even though it is tried and tested – this paradigm is really lacking in security. And even if organisations enforce secure password policies and users adhere to them, they may still be at risk. Recent developments in password cracking such as using GPU-based cracking systems make the security of any password under a certain length questionable [45]. With the increasing value that online identities have (how would you feel if your GMail, your FaceBook or your Twitter account got compromised and someone reads your private data or tries to impersonate you?) we, as authors, are of the opinion that two-factor authentication should become much more common than it is now.

As the classification shows, to get rock solid security using two-factor authentication we feel that a real purpose-built hardware token should be used. Nevertheless, emerging solutions that rely on mobile phones as personal devices, such as OTP Apps, show great promise. If implemented properly, these solutions can add significant value security-wise.

There are three key problems currently inhibiting wide-scale deployment of two-factor authentication outside of the corporate and banking environment. The first is cost; OTP and PKI tokens are expensive (there are exceptions: interestingly, one of the largest deployments of OTP tokens is for online World-of-Warcraft [41]). The second is dependence on bespoke hard- and software. Especially PKI tokens suffer from the problem that they require the end-user to install driver software and security middleware that is not always available for all end-user platforms.

Finally, the last problem is the lack of adherence to open standards. This not only stops people integrating support for two-factor authentication into their online services, it also means that many two-factor products are single purpose only (e.g. a token issued by a bank cannot be used to authenticate for other services).

We have tried to let these three problems be reflected in the classification given in Table 1.

5 tiqr: EXAMPLE OF AN OPEN APPROACH

5.1 INTRODUCTION

In 2009 we experimented with Mobile PKI (see also §2.3.3) as a means of authentication. As the report [8] of our experiment shows, we were very happy with the results. The technology is user-friendly, very secure and – because of the open standards it is based on – easy to integrate.

The only major hurdle we encountered is the dependence on mobile operators. These operators are very hesitant about deploying the technology because it requires a SIM swap (most SIMs deployed in The Netherlands are not PKI capable), and because they do not feel that there is a strong business case to deploy the technology in terms of potential revenue from it.

As operator of the National Research and Education Network (NREN) in The Netherlands, SURFnet operates a so-called identity federation (see [29]) called *SURFfederatie*. This federation enables users to log in at a multitude of online service providers using a single identity hosted by their home institution. Furthermore, this federation offers users single sign-on.

As is the case on most of the Internet, almost all authentications in the *SURFfederatie* rely on the tried and tested username/password mechanism. We would like to improve on this situation by introducing alternative means of authentication based on two-factor authentication technology. There are two reasons for this: first, we feel that some services require a stronger form of authentication than username/password. Secondly, we would like to offer users a safe alternative that they can use on untrusted systems such as, for instance, computers in Internet cafés.

SURFfederatie has a sizable and very heterogeneous user population consisting of approximately one million students, researchers and other staff from over a 160 different institutions. It would be impossible to deploy a token-based two-factor authentication solution because of the logistics involved. It would, however, be ideal if we could deploy a secure two-factor authentication system that uses mobile phones. Almost everyone owns a mobile phone (in fact, in The Netherlands, a country of 16.5 million people, there are over 19 million active mobile subscriptions [30]) and users are very motivated to carry their mobile phone at all times [31].

For reasons mentioned before, we could not rely on Mobile PKI so we started searching for an alternative. The criteria for this alternative were that it should be secure, user-friendly, easy to deploy, open and suitable for managing multiple identities. We believe that we have developed a novel solution that meets all of these criteria.

5.2 THE CONCEPT

5.2.1 BASIC FEATURES USED

The concept we call tiqr is based on three features of modern smartphones:

- The ability to run Apps
- A camera
- Internet connectivity

5.2.2 QR CODES

Relying on these smartphone features allows tiqr to make use of two-dimensional barcodes called QR codes. They were invented by Toyota subsidiary Denso-Wave in the 1990s.

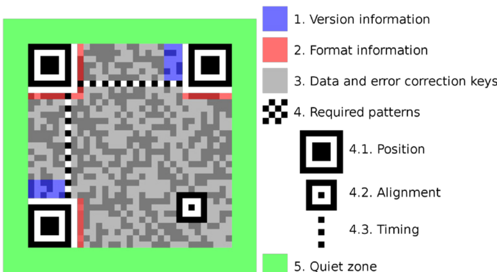


Figure 3 - a QR code with specific features highlighted (source [9])

Although patented, QR codes can be used royalty free. The technology behind the codes has been standardised as ISO/IEC 18004:2006. Up to 4KB of alphanumeric data can be stored in the codes and numerous libraries are available that can extract information contained in a QR code from images captured by a camera. For more details about QR codes, we refer readers to the excellent Wikipedia article [9].

QR codes have become quite popular, because most phones are equipped with cameras and can run QR code reader software. The codes are almost exclusively used in a static fashion, for instance in advertising or on public transport stops. They usually contain an encoded URL that QR code readers can open in a mobile browser.

The innovation we have come up with is to use QR codes in a dynamic rather than a static fashion. By encoding a challenge in a dynamically generated QR code that is displayed to the user when he/she wants to log in, we use QR codes to take away the burden on users of typing challenge/response codes. QR codes are also used during enrolment to tie the user's phone to an identity. Although this solution is not unique – the Google Authenticator App [28] can use a QR code to convey the user secret during enrolment – we have taken this technology further by creating a seamless user experience.

5.2.3 THE TIQR USER EXPERIENCE

To illustrate how tiqr works, we will go through the tiqr user experience during authentication (assume for now that a user already has a tiqr-enabled account).

The flow starts by a user surfing to a website that requires them to log in. Where most sites would display a username/password dialog (or an entry field to enter a one-time password), with tiqr users will see a QR tag as shown in Figure 4.

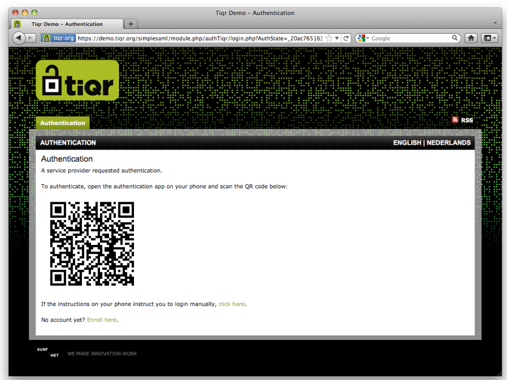


Figure 4 - tiqr login page showing a QR code

Contained in the QR code is a challenge. The user now launches the tiqr App on their smartphone. The App will activate the camera allowing the user to scan the QR code.

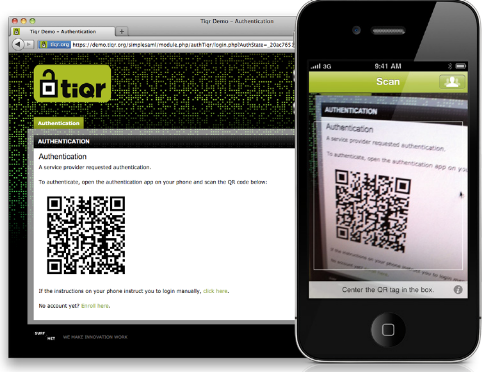


Figure 5 - the user scans the QR code with the tiqr App

Apart from a random challenge, the QR code also contains information on the relying party requesting authentication. The App can manage multiple identities and will select an appropriate identity that can be used to log in to this particular site (if multiple identities are present, the user will see a list and can choose the appropriate one). The tiqr App now asks the user to confirm that he/she wants to log in, also displaying the domain name of the site they are logging in to in order to reduce the risk of phishing.

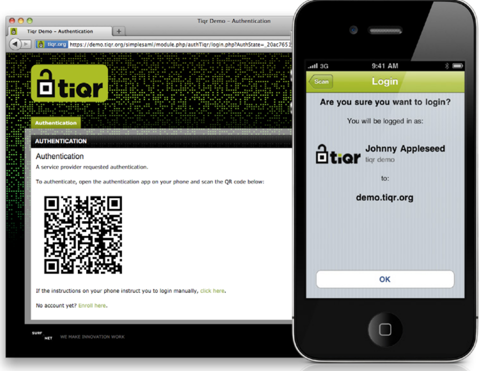


Figure 6 - tiqr asks for user confirmation

Once the user has confirmed their identity, they will be asked to enter their PIN code (the second factor).

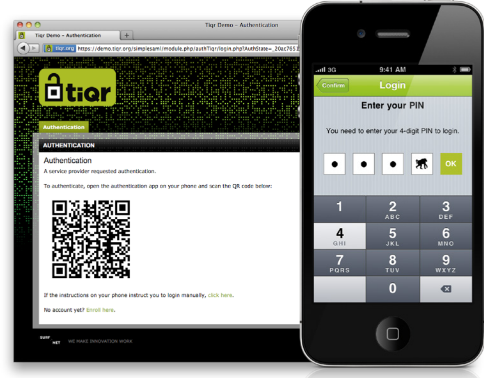


Figure 7 - user entering their PIN

The user is helped in remembering his or her PIN by means of animal icons displayed in the PIN entry dialog. Errors made during PIN entry (such as swapping two digits or a completely different PIN) will lead to a different sequence being displayed. When the user presses OK, login will proceed. If the user's phone is online, the Internet connection of the phone will be used to submit the response to the authenticating server thus obviating the need to type one-time passwords in to a website. When authentication is successful, the user is notified both on the phone as well as by the website proceeding with login by redirecting the user to the protected content as shown in the screenshot (Figure 8).

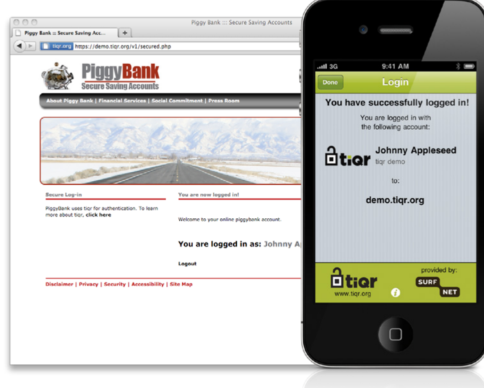


Figure 8 - the user has successfully logged in

In case no Internet connection is available on the phone a fall-back scenario is used where a one-time password is displayed on the phone for the user to type into the website (more on this in section 5.8).

5.2.4 FROM PROOF-OF-CONCEPT TO PRODUCT

We first came up with the concept that led to the development of tiqr in September 2010. In order to prove that the concept would work, we designed the initial protocol and developed a proof-of-concept implementation, both of the server side as well as of the phone side. For the proof-of-concept an implementation was created for Apple's iOS platform.

The proof-of-concept quickly showed that the technology worked very well. We first demonstrated the working proof-of-concept at an event held every two years to showcase SURFnet innovations to our connected institutions in December 2010 and received helpful and positive feedback from the people attending. This led us to decide that we should continue development.

In April 2011 we released the first Apple iOS production version in the Apple App Store and we presented on the project at the Internet2 Spring Member Meeting in Arlington, VA. The Android version was released in May 2011 just before we presented on further improvements to tiqr at the TERENA Networking Conference 2011 in Prague, Czech Republic.

The remainder of this section will go into more detail about the tiqr technology.

5.3 MOBILE APPS

5.3.1 PLATFORMS

We wanted to make tiqr available on the two most common smart phone platforms. According to a Q1 2011 market survey, those platforms are Apple's iOS and Google's Android platform:

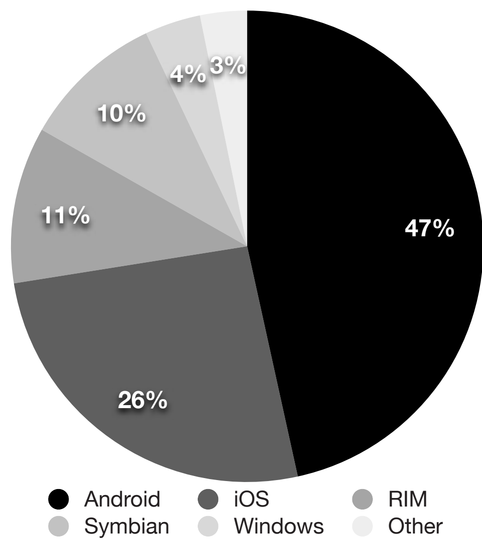


Figure 9 - smart phone market, source: The Guardian/Kantar

We have developed Apps for both these platforms. The Apps rely on the excellent ZXing QR code library developed by Google (see [18]) for QR code detection and decoding. The Apps implement the tiqr challenge/response protocol, which is based on OCRA/HOTP [19], [2]; more information on the protocol can be found in section 5.5.

5.3.2 APP SECURITY CONSIDERATIONS

The tiqr protocol relies on shared secrets for the challenge/response implementation. The secret is stored both on the phone as well as on the server.

We can only reasonably assume that the phone with the App and the secret on it is a secure authentication factor if it is hard for an attacker to gain access to the actual secret. We therefore protect the secrets belonging to user identities by encrypting the secrets using PKCS #5 password-based encryption [20]. The basis for encryption is the 4-digit PIN code the user chooses for the identity.

Of course there are only 10000 possible PIN codes with a 4-digit PIN. We assume that it is easy for a motivated attacker to gain access to the encrypted secret so we need to protect it against brute-force attacks. We achieve this by applying two principles. Firstly, the encrypted secret contains no internal structure (i.e. only the secret key – which is assumed to be truly random – is encrypted, there is no formatting around the key data before it is encrypted). This automatically leads to a second level of protection: because the encrypted key has no structure around it, it is impossible to check if the

correct PIN was used to decrypt the secret since the decrypted data will look like random data in all cases. As a result of this, only the server can check if the correct PIN was entered because the computed response is only valid if the correct secret key was used.

To prevent online attacks, we recommend that the server block an account after a pre-set number of failed authentication attempts (in fact our demo implementation blocks an account after 3 failed attempts). Depending on the desired security level, the server administrator may also decide to implement some form of exponential back off mechanism to mitigate brute-force attacks. In this scenario, accounts are temporarily blocked after a failed login attempt. This thwarts brute-force attacks but is also more user friendly for legitimate users since entering the wrong PIN more than a certain number of times will not immediately lead to a blocked account.

5.3.3 APP USER EXPERIENCE

One of our main goals was to create an easy-to-use system. We have taken special care to ensure that the user experience of the App is as straightforward, self-explanatory and smooth as possible. The prototype developed for the proof-of-concept was handed over to user-interface designers. They studied the concept and the prototype implementation. Using storyboards, they designed an optimised user workflow. The main focus of the workflow is to make it self-evident to the user what the next logical step is going to be. Another change they introduced was to do away with a separate enrolment workflow (in the prototype, we had two completely separate workflows for enrolment and authentication). Instead, the user just scans the QR code that is shown and information in the code determines whether an authentication or an enrolment workflow is going to be followed.

Another design decision that was made was to try to steer users toward using the same PIN code for all the identities managed by the tiqr App. The reasoning behind this is that the user-interface designers feel that it is counter-intuitive for most users to have multiple PIN codes in a single application. This concept is on the one hand very subtly integrated in the user experience by using suggestive wording (i.e. when enroling a new identity using the text "Please enter *your* PIN" when they have to choose a new PIN for the identity rather than "Please choose a new PIN"). On the other hand, it has also been taken quite far in that if a single identity becomes blocked due to entering the wrong PIN too many times, the App will block all identities it manages. We have not had sufficient user feedback to be able to decide whether or not this was a good choice; so far, we have had some feedback from third party developers that they feel this to be a bad choice. We hope to learn more in a pilot implementation that we are planning for the fall of 2011.

One final thing to note about the user experience is that we integrated an *aide-memoire* into the PIN entry dialog. We use icons with animal shapes to help the user remember their PIN (as shown in the figure below).

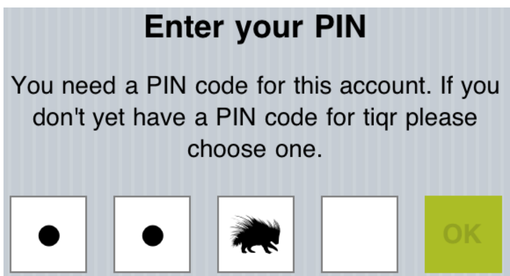


Figure 10 - PIN entry showing animal reminders

If the user enters the correct PIN, the same four animal icons should show up in the PIN entry field. Users can either remember the whole sequence or elect to remember just the last icon. To ensure that the sequence changes when common PIN entry mistakes are made (such as swapping two digits) we use the Verhoeff checksum algorithm [32] for error detection.

5.4 SERVER SIDE

5.4.1 REQUIREMENTS

As was already mentioned in the previous section, the basis for tiqr is challenge/response authentication using shared secrets (more information on the protocol can be found in section 5.5). This means that the secret key information that is present on the phone also needs to be stored on the server.

This, of course, puts certain requirements on the server implementation. User secrets should be stored encrypted, either on disk in a database or in a Hardware Security Module (HSM).

Another thing that is required on the server side is a library that generates the QR codes used to convey the challenge to the user. There are good open source implementations available for most common web application platforms. For our reference implementation we use PHP QR Code [33].

The most important thing to pay attention to on the server side is that the protocol is implemented correctly. We provide a reference implementation to show how the protocol works (which is discussed in the next section).

5.4.2 REFERENCE IMPLEMENTATION

To give developers a head start at integrating tiqr into their application, we have developed a reference implementation in PHP. This reference im-

plementation shows how the tiqr protocol works (see section 5.5).

We did not put any security provisions in the reference implementation so it should not be used in production. We are considering creating a more secure implementation that people can deploy straightaway.

5.4.3 SIMPLESAMLPHP MODULE

As outlined in section 5.1, we plan to use tiqr in an identity federation. To show this concept in action, we have developed a plug-in module for the popular SimpleSAMLphp [34] identity management suite.

Our demo portal (https://tiqr.org/demo/) uses this implementation.

It is currently based on our reference implementation so it is not sufficiently secure yet for production use. We are collaborating with the SimpleSAMLphp team to create a production-ready version, which we hope to release in the autumn of 2011.

5.5 PROTOCOL

5.5.1 GENERAL

We rely solely on open standards and open specifications as a basis for the tiqr protocol. The following standards are used:

- OCRA [19] – this is the suite of one-time password algorithms used for tiqr challenge/response
- JSON [35] – this is the object notation used to exchange data in the tiqr protocol
- HTTP over TLS – used to transport information exchanges securely
- QR codes [9]

5.5.2 ENROLMENT

Enrolment starts with a QR code that is displayed to the user. This QR code contains a URL with the following schema:

```
tiqrenroll://<url>
```

Where *<url>* must be a valid HTTPS URL that points to a location where the details for the enrolment request can be retrieved, for example:

```
tiqrenroll://https://demo.tiqr.org/enroll
/details?session=082176122169132630
```

The tiqr App will contact this URL to retrieve a JSON object with enrolment details. This object has the following syntax:

```
{
    "service": {
        "identifier": <id>,
        "displayName": <name>,
        "logoUrl": <logo-url>,
        "infoUrl": <info-url>,
        "authenticationUrl": <auth-url>,
        "ocraSuite": <OCRA-suite>,
        "enrollmentUrl": <enroll-url>,
    },
    "identity": {
        "identifier": <uid>,
        "displayName": <fullName>
    }
}
```

The *service* section of the object identifies the service to which the user is enrolling. The *identity* section provides details about the identity that is being enrolled. The fields in both sections of this object have the following semantics:

- Service section
  - *identifier* – should contain a reversed domain name (e.g. org.tiqr.demo)
  - *displayName* – should contain the name of the service
  - *logoUrl* – should contain a valid URL to a service logo; we recommend a PNG24 image
  - *infoUrl* – a URL linking to a webpage with more information about the identity provider; this link is displayed on the "detailed information page" for the identity
  - *authenticationUrl* – should contain the URL for the authentication handler for this service
  - *ocraSuite* – the OCRA suite the server requires; the App uses this to determine the appropriate OCRA parameters (see [19], section 6)*
  - *enrollmentUrl* – should contain the URL for the one-time enrolment handler
- Identity section
  - *identifier* – should contain a unique user identifier used to identify the account
  - *displayName* – should contain the full name of the user

*An example OCRA suite as specified by the server could for instance be:

```
OCRA-1:HOTP-SHA1-6:QH10-S
```

This OCRA suite specification breaks down as follows:

- *OCRA-1* – the OCRA algorithm version (in this case version 1, the current version)
- *HOTP-SHA1-6* – the cryptographic function to use (in this case HMAC OTP [2], with SHA-1 as hash algorithm using dynamic truncation to a 6-digit value); the tiqr App supports all algorithms specified in the OCRA standard
- *QH10-S* – the input for the challenge (in this case a 10-digit hexadecimal value represented as a string) and the size of the session data (in this case the default value of 64 bytes)

For more examples see [19].

When the user confirms enrolment, a new HTTPS connection is made to the enrolment server URL specified in the JSON object *enrollmentUrl* property. A POST request is sent across this link. This POST contains the following parameters:

- *secret* – this is the shared secret; the secret is generated by the App on the phone; we currently use 256-bit AES keys as secrets
- *notificationType* – optional; this is the notification type used to send push messages to the App and can be set to either APNS (for Apple Push Notification Service) or C2DM (for Android push notifications)
- *notificationAddress* – optional; notification-protocol specific address to which push notifications can be sent
- *language* – contains the user interface language of the user; this information may be used to display appropriate error messages in the user's preferred language

If enrolment is successful, the server will return the string *OK* (with no white space before or after the string). When an error occurs, the normal HTTP error procedure is followed to return the error to the App.

5.5.3 AUTHENTICATION

Authentication starts by displaying a QR code to the user. This QR code contains a URL encoded according to the following URL schema:

```
tiqrauth://[<identityIdentifier>@]
     <serviceIdentifier>/
     <sessionKey>/
     <challenge>[?<return Url>]
```

The fields in this URL have the following semantics:

- *identityIdentifier* – optional field specifying the user identity to use for authentication; may be used in a so-called step-up authentication scenario where the user has already logged in using another means of authentication
- *serviceIdentifier* – the service identifier as specified during enrolment (the service domain name in reverse domain notation, e.g. org.tiqr.demo)
- *sessionKey* – session key for this authentication request; links the response to the active user session when submitted
- *challenge* – the authentication challenge; the size of the challenge depends on the OCRA suite as specified during enrolment
- *returnUrl* – optional field specifying the URL to return the user to after successful authentication; this URL is only used if the session originated from the mobile browser on the device containing the tiqr App

The tiqr App will compute the response to the challenge using the algorithm that was specified during enrolment. It will submit the response by setting up a HTTPS connection to the authentication endpoint specified during enrolment. The submission is done using a POST with the following parameters:

- *sessionKey* – the session key received in the QR code identifying the user session that requires authentication
- *userId* – the user identifier of the user attempting to log in
- *response* – the response computed to the challenge specified in the QR code
- *language* – the user's preferred language; this information is used to display error messages in an appropriate language

If authentication was successful, the POST request returns the string *OK* (with no white space preceding or following the string). If authentication fails, the server will return one of the following error messages:

- *INVALID_RESPONSE[:attemptsLeft]* – the response provided to the challenge was invalid; this is interpreted by both the App as well as the server as an incorrect PIN entry. The optional integer value *attemptsLeft* indicates the number of tries left to return a correct response (and enter the correct PIN)
- *INVALID_USERID* – the server does not know the specified user
- *INVALID_CHALLENGE* – there is no known challenge for the current session; this usually indicates that the challenge has become invalid because of a timeout
- *ACCOUNT_BLOCKED[:seconds]* – indicates that the response provided to the challenge was invalid and that the associated user account is now blocked on the server; optionally, the account may be temporarily blocked for a specified number of seconds (this feature will be implemented as of version 1.2 of the tiqr App)
- *INVALID_REQUEST* – the POST request contained incorrect parameter data and was not accepted by the server
- *ERROR* – an unspecified error occurred

5.6 INTEGRATION WITH APPLICATIONS

As we already mentioned in section 5.4.2 and 5.4.3, we already provide several options for integrating tiqr into existing applications. The reference implementations we provide can serve as a basis for integration into web applications, but tiqr can also be used in other contexts.

Shortly after the first release of tiqr an independent software vendor, RCDevs from France, integrated support for tiqr into their OpenOTP Authentication Server [36]. Based on their existing integration with several products they were able to show that tiqr can – for instance – be used as an

authentication method to log users in to a secure shell (SSH) session (see Figure 11 for an example).

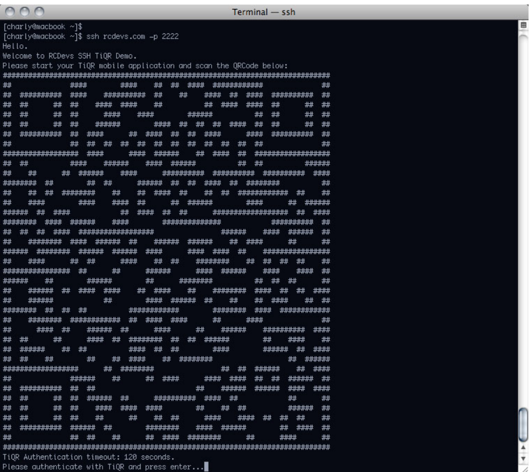


Figure 11 - using tiqr to authenticate an SSH session*

*Screenshot courtesy of Charly Rohart from RCDevs

Another example of integration into third party frameworks is the open request to integrate tiqr support into Shibboleth [37], a much-used framework for federated identity management.

5.7 SECURITY AUDIT

We hired an independent security auditor – Eindhoven-based Madison Gurkha, see http://www.madisongurkha.nl/ – to assess the security of tiqr. The goals we set them were to:

- Assess the architecture and design of tiqr from a security perspective
- Perform a code audit of both the App for iOS as well as for Android
- Perform a code audit of the reference server-side implementation
- Perform security tests on the live solution (both server as well as client side)

The security audit was performed on version 1.0 of the App (as released in April 2011 for iOS and May 2011 for Android) and was finished in June. The outcome of the security audit was positive; although the auditors identified several issues that needed resolving, they did not find any flaws in the architecture of the solution (note: the audit report will be published on the tiqr website, https://tiqr.org/, in the autumn of 2011). The most important remark from the auditors was that – strictly speaking – tiqr does not offer full two-factor authentication since the smart phone platform is much more accessible to evildoers than say a purpose-built hardware OTP token. We agree with this but would like to add that tiqr nevertheless is a vast improvement security-wise over username/password. And relying on a smart phone also has distinct advantages; recent research has shown that users are likely to notice that their phone is missing fairly quickly (see [44]). We think that this is much less likely to be the case for e.g. OTP tokens, since the single-purpose nature of these devices means they are used much less frequently.

We have taken care to include several security measures to deal with the inherent untrustworthy nature of smart phone platforms (see 5.3.2). The auditors agree that these measures indeed significantly enhance the security and they also agree that tiqr is an attractive and more secure alternative to username/password. They caution though that it is not fully equivalent to the security a hardware OTP token can offer. We would like to note that this is also true for the more traditional OTP Apps that OTP token vendors have started offering (see 2.4.2).

Another remark that the auditors made is that tiqr is potentially vulnerable to phishing. Attackers could perform a man-in-the-middle attack by initiating an authentication session, thus retrieving the QR code containing the challenge and by displaying this on a fake site, tricking the user in to logging in but instead giving the attacker access to their account. We agree that this is a risk and as mitigation the tiqr App always displays the fully qualified domain name of the site that the user is being authenticated to. Users are expected to validate the authenticity of the site they are logging into in the same way they do for e.g. their banking site, i.e. by checking the site's URL and server certificate. Note that this problem is not unique to tiqr; all OTP solutions are equally vulnerable to phishing.

The issues in the code and design identified by the auditors were resolved in version 1.1 of the App and in the reference server-side implementation that is available from the tiqr website. We also contributed fixes for the vulnerabilities in the OCRA reference implementation back to the authors of the RFC.

5.8 AVAILABILITY

From the onset it has been our goal to make tiqr freely available to all Internet users. To achieve this goal, we have released all relevant software in open source under a BSD-style licence and we have made the tiqr Apps available for free in both the App Store as well as on the Android Market.

All source code and documentation as well as a demo server can be found on our website, https://tiqr.org/

5.9 ROADMAP AND FUTURE WORK

Now that a production-ready version is available our next step will be to deploy tiqr within our own organisation. SURFnet currently uses X.509 software certificates for authentication to certain services. We plan to replace these by tiqr. All SURFnet employees have either an iOS- or Android-

based smart phone making it an ideal user population in which to use tiqr.

When we have gained real-world experience we will evaluate this deployment. Then, we plan to gradually introduce tiqr as an alternative means of authentication (alternative to username/password and/or SMS authentication) in some of the services we offer to our constituency.

We are also talking to our connected institutions to set up a pilot with a larger population. Our goal is to provide tiqr as an alternative means of authentication on an identity provider in our federation who currently only offer username/password.

One of the things we will be evaluating in these pilot deployments is whether or not the paradigm of encouraging users to use the same PIN for all their tiqr accounts works and whether or not it makes sense to block all accounts if one account needs to be blocked because of too many failed attempts at entering the correct PIN.

From a technological perspective, we are considering pursuing several areas of research:

- Turning tiqr into a true hardware token by leveraging the possibilities offered by SD cards with an embedded smart card controller (smartSD cards, see [39])
- Incorporating attribute release into tiqr (where tiqr releases attributes about a user asserted by a trusted third party), similar to the InfoCards paradigm (see [38])
- Using advances in cryptography such as zero-knowledge proof to further enhance the privacy aspects of tiqr.
- Using tiqr for transaction signing (as is e.g. done by banks with OTP tokens to approve financial transactions).

Some of these we will probably do ourselves, others we hope to pursue together with the academic community in the areas of cryptography and digital security.

5.10 REFLECTION

When we started the tiqr project we set out to create a two-factor authentication solution that would leverage the benefits of using a device that (almost) everybody has: a mobile phone. We were also mindful that the solution should be an improvement over username/password given the criteria introduced in section 4 and if possible it should also offer advantages over more traditional solutions.

We feel that with tiqr we have achieved most of these goals. We believe tiqr to be very user-friendly (much more so than some other two factor authentication solutions) and also believe that tiqr is an improvement in terms of security over username/password and on a par in that respect with many other two-factor authentication solutions. Furthermore, we believe that tiqr can be a viable replacement for more traditional OTP solutions, especially the OTP Apps and SMS authentication.

We feel that we should point out, though, that tiqr is not a panacea that solves all problems in (two-factor) authentication. It is, for instance, just as vulnerable to phishing as traditional OTP solutions. And because it does not rely on purpose-built hardware to store the secret data associated with a user's identity its security is not as strong as traditional tokens.

Nevertheless, we feel that tiqr is a useful addition to the two-factor authentication landscape. Its user-friendliness and the control over deployment it gives to organisations are also strong points.

6 CLASSIFICATION REVISITED

6.1 INTRODUCTION

In section 4 we introduced a classification for authentication solutions, judging solutions on hardware (in)-dependence, software (in)-dependence, security, cost, open standards compliance and ease-of-use. Now that we have introduced tiqr, we will revisit this classification.

6.2 CLASSIFICATION OF TIQR

Table 1 showed a classification of authentication solutions according to the criteria introduced in section 4. In Table 2 we have reprinted this classification and added tiqr at the bottom of the table (marked in grey).

	Hardware indep.	Software indep.	Security	Cost	Open Standards	Ease-of-use
Usern./pwd	++	++	--	++	=	+/-
OTP token	-	-	++	--	-/=	+
C/R token	-	-	++	--	-/=	+
PKI token	--	--	++	--	=	+
Mobile PKI	+	+	++	?	+	++
SMS OTP	+	=	-	-	--	-
OTP Apps	+	+/=	+	+/=	+/=	=
tiqr	+/=	+/=	+	+	++	++

Table 2 - Classification including tiqr

Again, one could argue that any classification is subjective, especially since we are judging our own solution. Therefore, we have tried to justify the classification we have assigned to tiqr below:

- *Hardware (in-)dependence* – tiqr requires advanced features only available on smart phones; it is therefore not as hardware independent as some of the other solutions that rely on mobile phones
- *Software (in-)dependence* – tiqr is currently only available for two smart phone platforms

- *Security* – although not as secure as a dedicated token, tiqr is much more secure than username/password and on a par with other OTP Apps
- *Cost* – tiqr is open source and available for free; the only inhibiting factor may be the cost of the device required to run tiqr
- *Open standards* – tiqr was built from the ground up to include open standards and the tiqr protocol itself has also been published
- *Ease-of-use* – tiqr was designed to be user friendly from the ground up by skilled interface designers

7 CONCLUSIONS AND RECOMMENDATIONS

7.1 THE NEED FOR TWO-FACTOR AUTHENTICATION

Our lives are increasingly being lived in the digital world. Social networks have become the staple of a new generation and many professionals cannot live without e-mail, VoIP, and services like LinkedIn. Governments and the public sector are also increasingly making vast amounts of often personal data (like medical records) available online.

This means that the value of the digital identities we use to access these services are becoming ever more valuable. It is no longer just your credit card that is at risk of being stolen, whole identities get hijacked.

We feel that it is inevitable that two-factor authentication becomes more widespread and actively try to stimulate its adoption, both within our own community as well as on a wider scale.

7.2 OPEN STANDARDS AND OPEN SOURCE

The best way forward to bring two-factor authentication to a wider audience is the adoption of open standards by vendors. This applies foremost to vendors of hardware OTP and PKI tokens. It is also of paramount importance that integration of two-factor authentication in online services becomes easier. One way of achieving this is by releasing open source solutions with flexible licenses. These can serve as useful examples and facilitate rapid integration.

7.3 TIQR

We have strived to practice what we preach when creating tiqr. We have focused on creating an open standards-based, open source and easy-to-use solution that is freely available. We believe that we have succeeded in the goals we set ourselves in that respect and we hope that tiqr can serve as a starting point for many organisations who want to integrate two-factor authentication into their online services.

What is also noteworthy to mention is that parts of the tiqr project have now been spun off as separate open source projects because of their general applicability. These include TokenExchange (an abstraction for push notifications supporting several device platforms), see [42], and a set of OCRA reference implementations in several programming languages, see [43].

7.4 RECOMMENDATIONS

We have already outlined recommendations for future work on tiqr in section 5.9. In addition to that, we would recommend any readers of this paper to invest some time into considering what two-factor authentication could add in terms of security both within their own organisations as well as for users of their online services.

8 ACKNOWLEDGEMENTS

The authors of this paper would like to thank:

- Ivo Jansch, Peter Verhage, Felix de Vliegher and Bas 't Hoen of Egeniq for their hard work implementing the mobile Apps and the demo server-side framework
- René Scheffer and Menno van de Laarschot of Stroomt for re-designing the user experience
- Charly Rohart of RCDevs for integrating tiqr in their software and developing the PAM module with tiqr support
- Petra Boezerooy of the SURFnet/Kennisnet subsidy programme for supporting the initial proof-of-concept that led to the development of tiqr

9 REFERENCES

[1] OATH, "VeriSign Introduces Collaborative Vision to Drive Ubiquitous Adoption of Strong Authentication Solutions", OATH website, February 2004, http://www.open authentication.org/news/040223

[2] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, O. Ranen, "HOTP: An HMAC-based One-T ime Password Algorithm", RFC 4226, The Internet Society, December 2005, http://tools.ietf.org/html/rfc4226

[3] D. M'Raihi, S. Machani, M. Pei, J. Rydell, "TOTP: Time-Based One-Time Password Algorithm", RFC 6238, The Internet Society, May 2011, http://tools.ietf.org/html/ rfc6238

[4] P. Hoyer, M. Pei, S. Machani, "Portable Symmetric Key Container (PSKC)", RFC 6030, The Internet Society, October 2010, http://tools.ietf.org/html/rfc6030

[5] R.M. van Rijswijk, M. Oostdijk, "Applications of Modern Cryptography", SURFnet, September 2010, http://www.surfnet.nl/documents/sn cryptoweb.pdf

[6] D. Raywood, "Google adds two-factor authentication to Gmail via SMS one time passwords",

SC Magazine UK, September 2010, http://www.scmagazineuk.com/googl e-adds-two-factor-authentication-to-gmail-via-sms-one-time-passwords/article/ 179266/
[7] A. Song, "Introducing Login Approvals", Facebook, May 2011, https://www.facebook.com/note.php?note_id=10150172618258920 &comments
[8] M. Oostdijk, M. Wegdam, "Mobile PKI, a technology scouting", Novay & SURFnet/Kennisnet, December 2009, http://www.terena.org/news /community/download.php?news_id=2528
[9] Wikipedia, "QR code", last visited June 2011, http://en.wikipedia.org/wiki/QR_code
[10] Vasco DIGIPASS GO range, last visited June 2011, http://www.vasco.com/products/digi pass/digipass_go_range/digipass_go.aspx
[11] RSA SecurID, last visited June 2011, http://www.rsa.com/node.aspx?id=1156
[12] Feitian OTP tokens, last visited June 2011, http://www.ftsafe.com/products/otp.html
[13] Vasco DIGIPASS Readers, last visited June 2011, http://www.vasco.com/products/digi pass/digipass_readers/digipass_readers.aspx
[14] SafeNet SafeWord GOLD, last visited June 2011, http://www.safenet-inc.com/uploaded Files/About_SafeNet/Resource_Library/Resou rce_Items/Product_Briefs_-_EDP/SafeNet_product_brief_GOLD.pdf
[15] A. Litan, "SMS/OTP under attack – Man in the Mobile", Gartner, September 2010, http://blogs.gartner.com/avivah-litan/2010 /09/28/smsotp-under-attack-man-in-the-mobile/
[16] J. Kaavi, "Strong authentication with mobile phones", Helsinki University of Technology, Fall 2010, http://www.cse.hut.fi/en/ publications/B/11/papers/kaavi.pdf
[17] "Specification for Integrated Circuit(s) Cards Interface Devices revision 1.1", DWG Smart-Card Integrated(s) Card Interface Devices, April 2005, http://www.usb.org /developers/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf
[18] "ZXing (Zebra Crossing) multi-format 1D/2D barcode image processing library", last visited June 2011, http://code.google.com/p/zxing/
[19] D. M'Raihi, J. Rydell, S. Bajaj, S. Machani, D. Naccache, "OCRA: OATH Challenge-Response Algorithms", Draft RFC, The Internet Society, March 2011, http://tools.ietf.org/html/draft-mraihi-mutual-oath-hotp-variants-14
[20] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification", RFC 2898, The Internet Society, September 2000, http:// tools.ietf.org/html/rfc2898
[21] "OpenSC – tools and libraries for smart cards", last visited June 2011, http://www.opensc-project.org/opensc
[22] "PKCS #11: Cryptographic Token Interface Standard", RSA Laboratories, June 2004, http://www.rsa.com/rsalabs/node.asp?id=21 33
[23] "ETSI TR 102 203 v1.1.1 – Mobile Signatures; Business and Functional Requirements", ETSI, May 2003, http://docbox.etsi.org/EC_Files/EC_Files/tr_102203v010101p.pd f
[24] "ETSI TR 102 204 v1.1.4 – Mobile Signature Service; Web Service Interface", ETSI, August 2003, http://docbox.etsi.org/EC_Fi les/EC_Files/ts_102204v010104p.pdf
[25] "ETSI TR 102 206 v1.1.3 – Mobile Signature Service; Security Framework", ETSI, August 2003, http://docbox.etsi.org/EC_Fi les/EC_Files/tr_102206v010103p.pdf
[26] "ETSI TR 102 207 v1.1.3 – Mobile Signature Service; Specifications for Roaming in Mobile Signature Services", ETSI, August 2003, http://docbox.etsi.org/EC_Files/EC_Fil es/ts_102207v010103p.pdf
[27] "YubiKey", last visited June 2011, http://www.yubico.com/yubikey
[28] "Installing Google Authenticator", last visited June 2011, http://www.google.com/ support/accounts/bin/answer.py?answer= 1066447
[29] "SURFfederatie – federated identity management for simpler cooperation and greater ease of use", SURFnet, 2007, http://www.surfnet.nl /Documents/attachment.db@189185.pdf
[30] "The Netherlands", CM International / CM Telecom, last visited June 2011, http://www.cmtelecom.com/premium-sms/netherlands
[31] R. Fergusson, "One third of mobile owners notice phone is missing within 15 minutes", Engineering & Technology Magazine, March 2011, http://eandt.theiet.org/news/2011/mar /securenvoy-survey.cfm
[32] N.R. Wagner, "The Laws of Cryptography: Verhoeff's Decimal Error Detection", University of Texas San Antonio, 2002, http://www.cs.utsa.edu/~wagner/laws/verho eff.html
[33] "PHP QR Code – QR code generator, an LGPL PHP library", last visited June 2011, http://phpqrcode.sourceforge.net/
[34] "SimpleSAMLphp", last visited June 2011, http://simplesamlphp.org/
[35] "JSON – JavaScript Object Notation", last visited June 2011, http://www.json.org/
[36] "TiQR Authentication Server", last visited July 2011, http://www.rcdevs.com/products/ tiqr/
[37] "Add tiqr Two-Factor Authentication Mechanism", Shibboleth JIRA, last visited July 2011, https://issues.shibboleth.net/jira/brow se/IDP-91
[38] "The Information Card Ecosystem - Information Cards", Information Card Foundation, last visited July 2011, http://information card. net/

[39] "smartSD", SD Association, last visited July 2011, https://www.sdcard.org/developers/tech/smartsd/

[40] "Introducing AlterEgo – 1.5 Factor Authentication for Web Apps", The Rocket Science Group, last visited August 2011, http://blog.mailchimp.com/introducing-alterego-1-5-factor-authentication-for-web-apps/

[41] D. Winder, "Blizzard introduces most powerful World of Warcraft weapon ever", June 2008, last visited August 2011, http://www.itwire.com/business-it-news/networking/19106-blizzard-introduces-most-powerful-world-of-warcraft-weapon-ever

[42] Egeniq, SURFnet, "TokenExchange for iPhone, iPad, Android and BlackBerry device tokens", last visited August 2011, http://code.google.com/p/tokenexchange/

[43] Egeniq, SURFnet, "Example implementations of the OATH OCRA algorithm in various languages", last visited August 2011, http://code.google.com/p/ocra-implementations/

[44] DarkReading, "National Survey Finds 1 in 3 Mobile Phone Owners Would Know They've Lost Their Phone Within 15 Minutes", last visited August 2011, http://www.darkreading.com/insider-threat/167801100/security/client-security/229400606/national-survey-finds-1-in-3-mobile-phone-owners-would-know-they-ve-lost-their-phone-within-15-minutes.html

[45] R. Boyd, J.L. Davis and C. Mastrangelo, "Teraflop Troubles: The Power of Graphics Processing Units May Threaten the World's Password Security System", Georgia Tech Research Institute, last visited August 2011, http://www.gtri.gatech.edu/casestudy/Teraflop-Troubles-Power-Graphics-Processing-Units-GPUs-Password-Security-System

Building Useful Security Infrastructure For Free

Brad Lhotsky <brad.lhotsky@gmail.com>
National Institutes on Health, National Institute on Aging, Intramural Research Program

Tags: security, Perl, database, open source, syslog-ng, postgresql, FISMA

Introduction

Working as a Security Engineer for a research program in the Federal government is a lot of fun, but incredibly challenging. Research, rightfully, receives the lion's share of funding, leaving very little for support services like IT and no funding for security specific activities. However, the burden of designing, implementing, analyzing, and reporting compliance to weighty government IT Security mandates like FISMA falls squarely on the IT section.

Our IT staff is less than 10 people. We provide Help Desk, Linux server administration, networking (switches, IDS, firewalls, NMS), SQL Databases, SMB file shares, programming support, training, and implement in-house applications for scientific research mostly in Perl and PHP for our institute of 700-900 users. We are also responsible for reporting compliance with Federal, Institutional, and Divisional mandates to our oversight.

In order to achieve all of this with a small staff, we've designed and implemented a lot of automation based on Open Source Software. We've learned how to leverage these tools to meet the needs of our institute and the requirements of those above us.

As a pragmatic group with very little free time, we focus on building security tools that provide daily operational value. We simply do not have the resources to implement controls for the sake of the controls themselves.

The More You Know

Most IT Security controls focus on first understanding your systems. A system in this sense is defined as the computers, people, and networks that work together to perform a task. In order to begin classifying, we need to know what we have and where. The first step was to rollout a comprehensive centralized logging infrastructure for our UNIX and Windows servers.

We chose to use syslog-ng as a basis for our centralized logging platform for one incredibly useful feature:

```
destination d_subscriptions {
      program("/usr/local/eris/bin/syslog-ng-service.pl");
};
```

This feature starts the program specified keeping a handle of that program's STDIN open to dispatch messages to based on the "`log {}`" definitions specified in the configuration file. This removes the startup overhead from the called program allowing the use of programs written in dynamic scripting languages which incur enormous startup penalties. It also ensures that the program end point is available while syslog-ng is running, meaning there's no additional program supervision necessary.

In order to facilitate rapid development of syslog based event correlators, we developed this program to convert the incoming syslog stream into a TCP based service that a script can connect to and "subscribe" to feeds of interest. For instance, the inventory application subscribes to dhcpd, MSWinEventLog, sshd, arpwatch, and smbd. This keeps the database size smaller and focussed on the events that prove most useful to operations.

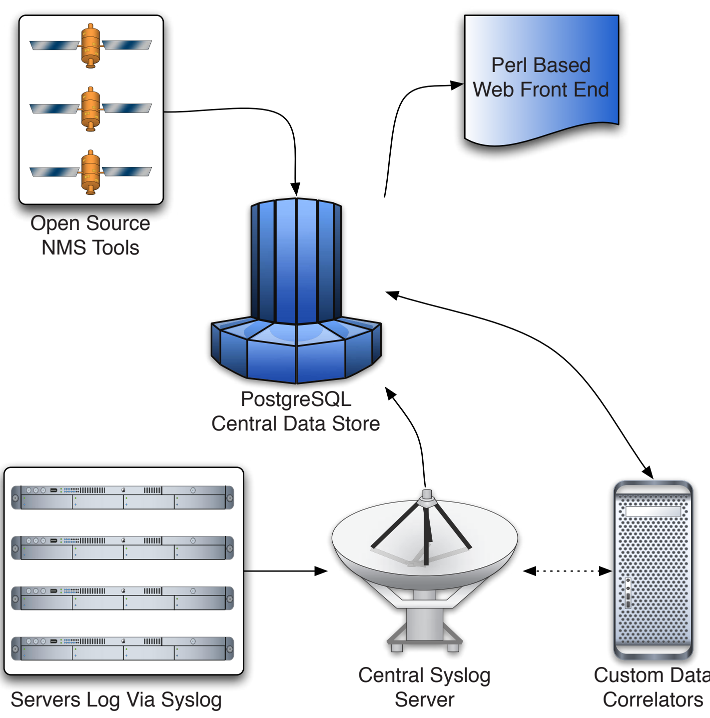
A Safe Place to Keep Our Data

In order to facilitate strange and novel concepts in correlation, it was clear that a Relational Database would be awesome as a storage engine to allow indexing and searching of the data we received. A PostgreSQL database server was setup and configured to allow data storage and retrieval. The reasons for this are numerous, but at the time of initial development it was the only Open Source database with views, stored procedures, triggers, and a slew of particularly relevant native data types including network types for IP addresses, networks, and MAC addresses. PostgreSQL has continued to make dramatic improvements to performance and usability since that time, and continues to be a leading Open Source RDBMS with unparalleled features and reliability.

PostgreSQL's PL/PgSQL language extension which was designed to be as close to Oracle's PL/SQL provides the option to do data correlation and validation in the database through the use of stored procedures and triggers. This facilitates rapid development of scripts placing data in the database as the "business logic" can be implemented at the data storage level. There is a performance penalty for doing this, but it allows for correlation to occur automatically as new datasources are added.

The setup we've designed is represented via the diagram on the following page.

Conceptual Overview of the System



Connecting the Dots

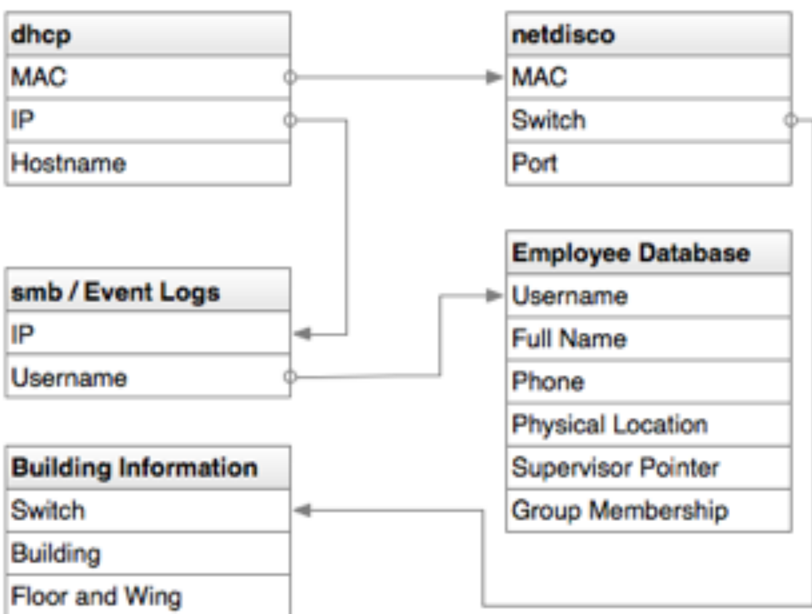
DHCP logs serve as a primary jumping off point for data correlation. We can simply store MAC, IP, and hostname attributes in a table and use them for lookups. We chose Netdisco as an Open Source layer 2 network management system that could be deployed to PostgreSQL. With a few triggers added to the Netdisco system, we can correlate MAC addresses to switch and port which allow our staff to quickly establish building, floor, and wing for any IP address on our network.

Using Samba and MS Windows Event Logs, we were able to discover the ActiveDirectory account name logged in to any client system. This allows simple IP to username correlation for things like IDS, but more importantly username to IP matching so our Help Desk don't have to walk every user through the "Start -> Run -> cmd -> ipconfig" routine, saving a few minutes with each call to the Help Desk.

Years prior to this system, our staff was asked to supplement the existing Enterprise Directory Service which was developed for all of NIH, with a number of specific enhancements specific to our Intramural Research Program. One of the features required every user to be assigned a "supervisor" attribute that links to another person object. Each person object contains full name, email, AD account, phone number, building, room, lab, and a pointer to the supervisor person object. This data was imported and synced to the PostgreSQL database as it would prove useful.

Basic Inventory Information

So with a carefully designed PG database and a few hundred lines of Perl code, we've managed to establish the following relationships:



What this means is:

- any event on the network containing a MAC address, IP address, or username can be correlated to any of the others
- and can also be correlated back to the metadata on the username and location

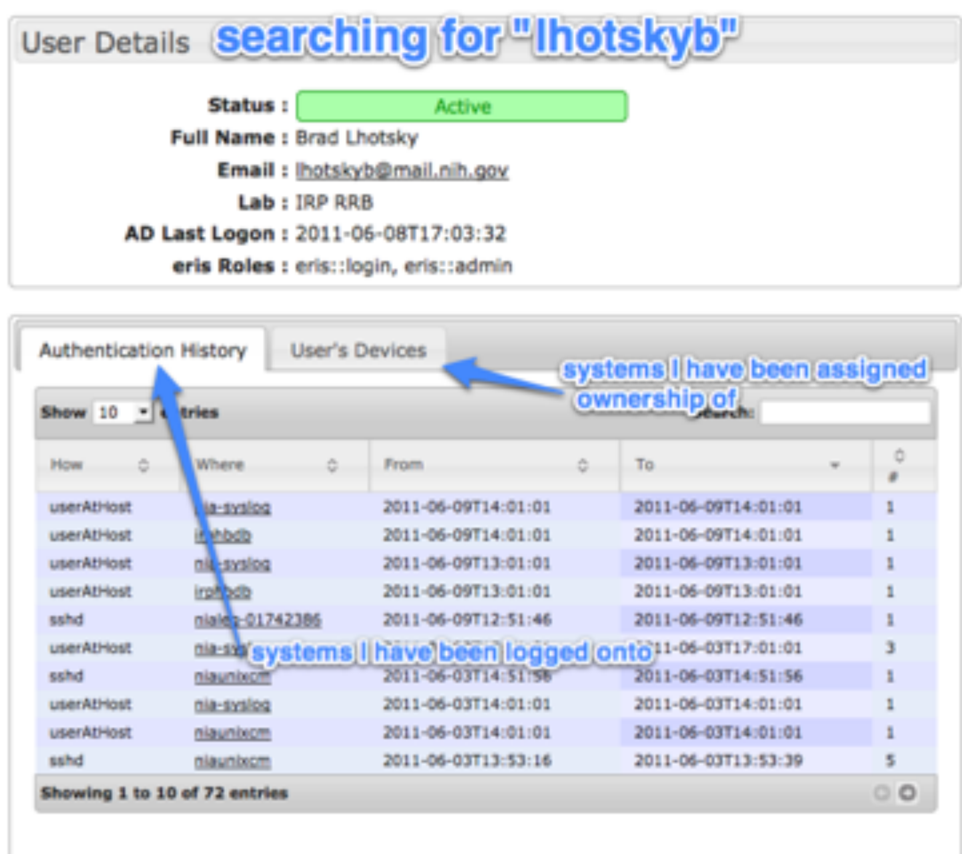
This allows classification of events in terms of business structure or geographic location all from data already available.

The tables which implement the storage keep track of dates and times inventory events such as DHCP, login, and ARP discovery occur. Since we're siphoning this data from the network servers and switches, we're not relying on high level or complicated protocol like IBM Tivoli End Point Manager to discover the devices. When a device is plugged into the network, we are able to immediately see it's been connected and record the date, time, and location of the event.

Now, when we need to report on the number of Apple computers, we can loop through the MAC Addresses seen in the past 6 months, look up the manufacturer data from the OUI Database, and report more accurately the number of active Apple Computers on our network.

Creating Useful Security

By wrapping this in a searchable web application, the Help Desk staff can now be far more efficient on each call.



This is excellent, but the real benefits of this system begin to show up as we integrated it with more Open Source security products.

Intrusion Detection and Correlation

We chose Snort as our Open Source IDS. Snort is free, fast, and stable. It does take a considerable amount of setup and configuration, but any signature based IDS solution will require that overhead. After being configured to listen through a network tap to a bonded interface on a CentOS box, we end up getting alerts that look like this:

```
Jun  2 12:10:55 myids snort[2908]: [1:2012647:2] ET POLICY Dropbox.com
Offsite File Backup in Use [Classification: Potential Corporate Privacy
Violation] [Priority: 1] {TCP} 137.x.x.x:1211 -> 199.47.216.144:80
```

This output is familiar to security professionals. We do have an IP address, which we have already established can be attributed back to a username, and a username back

to their the organizational unit. This alert is parsed by the system and the relevant data stripped off and classified. An alert like this is classified in our system as "Potential Data Loss Event."

We can then generate reports based on organizational unit and event classification which can tell us interesting things about a lab or section that we may not really want to know!

Configuration Management or DevOps for Compliance

All you base are belong to ..

Puppet, or any other configuration management engine that suits your needs will be sufficient. Spend some time evaluating the various configuration management engines and choose the one that best suits your organization. I cannot say enough good things about a good CM system that fits your organization.

We deployed Puppet and then put everything into our Version Control System (VCS) with commit hooks to automatically deploy new tagged release to the PuppetMaster. Using Puppet's Domain Specific Language (DSL), I was able to convince our small staff of old school developers to embrace VCS after I built and demonstrated this:

```
subversion::deploy { 'project_name':
      svnurl => 'svn+ssh://svn-readonly/repos/section/projectname',
      target => '/opt/local/project_name',
      notify => Service['httpd']
}
```

Which requires only a Subversion project directory with a trunk/ and tags/ subdirectory. A bash script for tagging releases is distributed to /usr/local/bin/svntag which makes creating incremental release tags as easy as typing "svntag." Puppet will then use $target/RELEASE to maintain the release number that's been deployed to that target and anytime a new release is tagged, it will be automatically deployed at that location. Additions for allowing hostname-based configuration files was incorporated into a macro based off this:

```
webapp::deploy { 'name': project => 'name', config => 'name.yml' }
```

This expands to doing much the same as the previous example including the httpd restart, but also deploys the application configuration after the checkout completes, overwriting the development configurations that are stored in the subversion repository.

We run RedHat based distributions (CentOS,Fedora, and now Scientific Linux). Puppet was a great start, but Cobbler has solidified our CM platform. Cobbler is a KickStart based systems build platform that utilizes PXE to automate installs of RedHat based distributions. There is some work in process to extend it's functionality to Debian

systems. One reason to choose a build system like Cobbler, instead of an imaging solution like Ghost, is deployment to a hodgepodge of hand-me-down hardware that we maintain in a Research program.

When Cobbler performs an install, it's configured to include Puppet in the build, setting it to run at first boot. Using Puppet and Cobbler to rebuild my IDS sensor when I had to replace the hard drive took 37 minutes from PXE boot to up and running with Snort and syslog-ng for centralized logging.

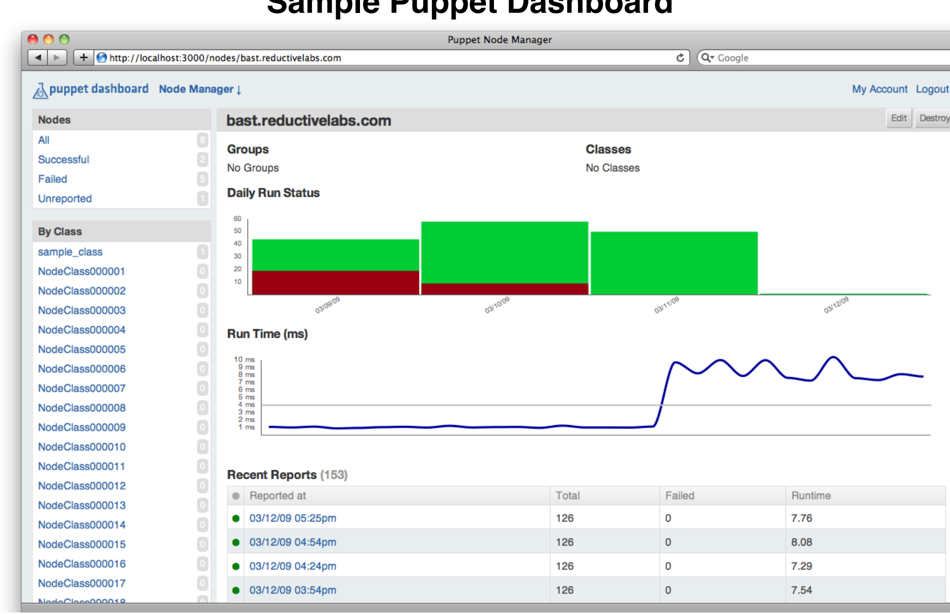
And what exactly does this have to do with Security?

A lot. Using a configuration management suite provides countless security benefits. First, it is the ultimate tool for guaranteeing consistent configuration across your network. It also offers the most benefit when each system is configured as much as possible by the CM. This allows a system administrator to PXE boot a new piece of hardware to replace an existing server and have the box configured identically in under an hour.

What we also get is a free inventory of all our servers. Since, as logical people, we tend to name classes and definitions something meaningful, we can leverage the CM tool to report on system functions and logical groupings. Puppet stores it's catalogs and states in simple YAML files which are parsed quickly and efficiently by your language of choice.

Configuration Managements, System Inventories, Software Inventories, are all provided. It's even possible to view the state of the compliance with the catalogs using the Puppet Dashboard. There are happy green and stressed out red lights for your auditors to admire!

Sample Puppet Dashboard



Extending the Functionality

After the collection and storage of all this data to a relational database, we've found it indispensable to solving problems on a day to day basis. From simple one-off scripts to determine the number of Apple computers on the network, to more extensive systems that were trivial to implement on the back of the data we've collected. Consider our researchers requirement to utilize Skype to collaborate with international colleagues at no cost. The Federal Government prohibits the use of Skype, unless there are adequate compensating controls in place.

Using the data we've collected, we developed an automated tracking of Skype users. To receive a waiver from the Departmental Policy we were required to Skype users at our Institute affirm a monthly "Rules of Behavior" (RoB) update. The process of discovery and tracking of Skype usage looks something like this:

1. IDS signatures classified as "Skype" are correlated to Usernames using the database.
2. The usernames are checked against a table of "Accepted RoB's" and compared.
   a. If this is the first event, ie, no rows in the RoB table, the user is emailed the RoB and must click a link, sign in, and agree to the terms
   b. Otherwise, the "last detected skype usage" timestamp is updated
3. If at the time of detection, it has been a month or more since the acceptance of the RoB, the user is again emailed the link, asked to sign in, and agree to the terms of the RoB.
4. Everyday a list of users required to accept the RoB is compounded and emailed to the Administrators with their status included.
   a. The administrators receive the Phone number, Building/Room information, and the Lab Manager details for each user, aiding in persuading the user to accept the RoB.

This system is mostly automated, except for the occasional phone calls to the users requesting that they agree to the RoB terms. Other potential solutions to this problem exist, but often require complex proxy configurations that break if the user takes their laptop offsite, or manual exceptions by statically assigned IP addresses and manual tracking of RoB Acceptance. Our solution saves time, energy, and resources. It is only minimally invasive to the end-users and barely noticeable to the administrators!

Lessons Learned

By choosing to develop this system in house, we have gained invaluable experience and knowledge. The development infrastructure to support the development of our custom inventory and security correlation engine has lead to near-mastery of Modern Perl, PostgreSQL, centralized logging infrastructure, and VCS, both Subversion and Git.

The fact that we're storing everything in a relational database provides us with the ability to "mash up" data from disparate sources. We've been able to successfully respond to data calls from our parent organization with SQL statements that we can reproduce time and time again.

Sure, we didn't get free t-shirts, calendars, and pens from vendors. We didn't go to training sessions at fancy hotels to learn to use each piece of the system. We can't hire someone with a specific vendor certification to replace a team member if they leave. However, the entire team has learned to work together and everyone has increased their abilities in many different areas. Our Help Desk staff know Windows, Linux, Mac OS X, and some rudimentary programming. They also understand the network and how it works. This type of training, which lacks the polish and formality offered by the big name vendors, is invaluable to day to day Operations.

But, the best part of the system is it's being used, every day,by Ops team members to make a difference.

Resources

Open Source Software Mentioned

- **syslog-ng**: Replacement for standard syslog
  - http://www.balabit.com/network-security/syslog-ng
- **PostgreSQL**: Open Source Relational Database
  - http://postgresql.org
- **Netdisco**: Open Source Network Management System
  - http://netdisco.org
- **Perl Components** (http://perl.com)
  - **Catalyst**: MVC Framework
    - http://catalyst.perl.org
  - **POE**: Event driven Perl library
    - http://poe.perl.org
- **Snort**: Open Source Intrusion Detection System
  - http://snort.org
- **Puppet**: Open Source Configuration Management Engine
  - http://www.puppetlabs.com
- **Subversion**: Open Source Centralized Version Control System
  - http://subversion.tigris.org/
- **Cobbler**: Open Source Installation Server
  - https://fedorahosted.org/cobbler/

Projects by Brad Lhotsky (https://github.com/reyjrar)

- **svnutils**: A collection of Subversion utilities for automatic deployment and integration with Puppet
  - https://github.com/reyjrar/svnutils
- **optperl**: Spec files for installing Perl into /opt including integration with Puppet
  - https://github.com/reyjrar/optperl
- **POE::Component::Client::eris**: Perl module for subscription based log tailing
  - https://github.com/reyjrar/POE-Component-Client-eris
- **eris**: Network Console which collects and correlates data
  - https://github.com/reyjrar/eris

Local System Security via SSHD Instrumentation

Scott Campbell
National Energy Research Scientific
Computing Center,
Lawrence Berkeley National Lab

scampbell@lbl.gov

ABSTRACT

In this paper we describe a method for near real-time identification of attack behavior and local security policy violations taking place over SSH. A rational is provided for the placement of instrumentation points within SSHD based on the analysis of data flow within the OpenSSH application as well as our overall architectural design and design principles. Sample attack and performance analysis examples are also provided.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information Flow Controls –

General Terms

Measurement, Security.

Keywords

SSH, keystroke logging, Bro IDS, Intrusion Detection, policy enforcement.

1. INTRODUCTION

The adoption of SSH as the defacto protocol for interactive shell access has proven to be extremely successful in terms of avoiding shared media credential theft and man in the middle attacks. At the same time it has also created difficulty for attack detection and forensic analysis for the computer security community. The SSH protocol and it's implementations such as OpenSSH [9] provide tremendous power and flexibility. Examples of this flexibility include authentication and encryption options, shell access, remote application execution and X11 and SOCKS forwarding. While the benefits gained vastly exceed the difficulties introduced by this protocol, the loss of visibility into user activity created problems for the security groups tasked with monitoring network based logins and activity.

The National Energy Research Scientific Computing Center (NERSC) is the primary open science computing facility for the Office of Science in the U.S. Department of Energy. It is one of the largest facilities in the world devoted to providing computational resources and expertise for basic scientific research, and has on the average 4000 users across seven primary computational platforms. The significant majority of user interaction involves interactive ssh logins. To address this lack of visibility into user activity on our high performance computing (HPC) infrastructure, we introduced an instrumentation layer into the OpenSSH application and feed the output into a real time analyzer based on the Bro IDS. This instrumentation provides application data such as user keystrokes and login details, as well as *metadata* from the SSHD such as session and channel creation details. This data is fed to an analyzer where local site security policy is applied to it, allowing decisions to be made regarding hostile activity. The data analyzer is based on the Bro intrusion detection system (IDS) [10] which provides a native scripting language to handle data structures, tables, timers to express local security policy. In this capacity Bro is being used as a flexible data interpreter. A key differentiator between the instrumented SSHD (iSSHD) and many other security tools and research projects is that iSSHD is not designed to detect and act on single anomalous events (like unexpected command sequences), but rather to enforce local security policy on data provided by the running SSHD instances.

A key idea is that the generation of data is completely decoupled from its analysis. The iSSHD instance generates data and the analyzer applies local policy to it. By using the Broccoli library [4], we convert the structured text data output by iSSHD into native bro events that are processed by the analyzer system [3]. Events, as their name implies, are single actions or decisions made by a user that are agnostic from a security analysis perspective. Bro processes these events in the same way as network traffic events, applying local security policy to interpret them as desired.

Local security policy can be thought of as sets of heuristics that describe (in this context) what behaviors are considered unacceptable or suspect. This behavior might be a command like "mkdir …", application usage like remotely executing a login shell, or tunneling traffic to avoid blocked ports. iSSHD was designed so that the installed SSHD instance would not need to be modified with every new threat. Instead, changes are made on the analysis/policy side as new problems are identified. This not only simplifies administration, but also allows experiments to be run on previous logs without significant work.

While NERSC has no explicit legal or privacy issues with intercepting communications on local systems, we recognize the importance of an informed user and staff population. To help address this we chose a policy of complete transparency. Each major group at NERSC was allowed representation in the design process and code review. As well, the entire user community was alerted to the changes by making announcements at User Group meetings and email notices. The complete source code is available to anyone interested and can be secured through the LBNL Technology Transfer Office.

The iSSHD project has been used in production capacity at NERSC for nearly three years on approximately 350 hosts. There are around 4000 user accounts with a daily average of 52,000 logins per day on the collective set of multi-user systems. In addition to the obvious security functionality, there are a number of other non-security purposes like debugging user problems or job analysis where having access to historical keystroke data has been quite beneficial in tracking down systems problems.

The reminder of the paper is structured as follows. In related work similar coding projects and tools are presented. Next the execution flow within an unmodified OpenSSH 5.8p1 instance is mapped out. This flow provides a way to determine the most effective points for instrumentation. In section four, the overall architecture and design goals are detailed including the integration of Bro into the process. Section five provides implementation

details describing the inherent tradeoffs between complete monitoring and resource limitations. Section six has examples of attacks and some rudimentary analysis. Finally future work and references are provided.

2. RELATED WORK

Related work can be generalized into several groups. These are research projects relating to SSH data access, hacker activities, and more generalized detection of SSH credential theft detection in the HPC environment.

The work most similar to our own involves the hacker community's use of backdoored SSHD instances to steal authentication credentials. In principle there is little difference between this behavior and the functionality provided by the iSSHD except in terms of the *breadth* of data provided. Statically backdoored OpenSSH code has been around since at least 1999 [14], and more recent versions are trivial to locate - see [15] for example.

Besides directly replacing the existing SSHD binary, there are at least three additional ways to access session data. The first is via direct access to a user's terminal devices by a privileged user. This can be achieved by one of dozens of small applications or as part of a larger kernel rootkit [18]. A more subtle approach is to interfere with kernel level behavior, thereby preventing a user space analysis of the terminals from giving away the access. Typically rather than just looking at terminal IO, input and output system calls are intercepted via a hidden kernel module. This information is transmitted to an analysis tool or recorded. There are innumerable examples of this approach within the rootkit community [11] as well as Honeypot implementations such as Sebek [13]. Finally you can interact with the running SSHD process by injecting code into it [16] or using process debugging to "jump" from their stolen user account to a potentially privileged session on another machine [17] [1]. These last two cases are somewhat subtle in that no changes to the actual static (non-running) binary are made.

There is a general class of SSH related security work focusing on user account theft via anomaly detection, both in terms of command sets as well as process accounting data. These include Yurcik [21] [22] and Joohan Lee et al. [5] who look for account compromises within the HPC domain via accounting and command analysis. Historically, there is a rich collection of research relating to account masquerading, with a nice write-up by Malek et al. [6]. This last class of ideas can be fed by or used with the iSSHD and incorporated into the sites overall intrusion detection design since they are orthogonal to the actual iSSHD.

3. SSH Application and Protocol

In order to identify the best places to place instrumentation within the SSH application, it is necessary to understand the code path taken by typical behavior as well as subtleties within the protocol.

From a historical perspective there are two individual (and incompatible) versions of the SSH protocol available. Tatu Ylönen created version 1 in 1995 as a replacement for the then ubiquitous telnet and rlogin protocols. OpenSSH emerged with the OpenBSD group taking up development after a number of organizational changes including the splitting of the Ylönen code base at one of it's last open source implementations. The SecSH IETF working group developed version 2 originally published in 1998 and in 2006 a revised version of the protocol was adopted as a standard in RFC 4250 (Protocol Assigned Numbers) [23], 4251 (Protocol Architecture) [24], 4252 (Authentication Protocol) [25], 4253 (Transport Layer Protocol) [26], 4254 (Connection Protocol) [27].

In terms of this analysis, all paths and descriptions assume the use of version 2 protocol since version 1 has suffered a number of pathological security defects [19] which reduce it's use to older and unusual cases. In the case of the actual code instrumentation, this assumption is not made and both version 1 and 2 provide nearly identical logging. Section 3.1 represents a general overview and relationship between RFC and OpenSSH structure. Section 3.2 takes this high level design and fleshes it out, providing a code path and rational for instrumentation locations.

3.1 SSH Application and Protocol Layering

For this initial description we avoid taking into consideration a number of details in order to focus on the overall flow of information and data. For a generic shell interaction a simplified diagram of the data flow might look something like Figure 1.

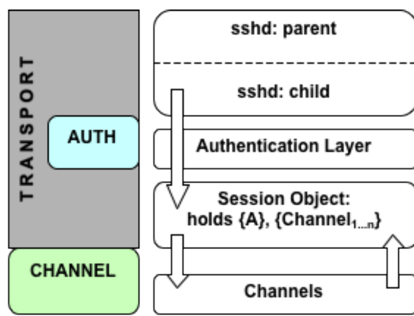


Figure 1: Application vs. Protocol design for typical SSHD session

Here Figure 1 is broken out into two columns – on the left there is the protocol layering as defined in RFC 4250-4254. The right side describes the application implementation of those layers. It is worth noting that the layers do not map 1 to 1 - in particular the role of the session object within the application, which according to RFC 4253 should be rolled into the transport layer. Here each application layer is a functional layer within the application, with the parent SSHD is represented as the top block. After a successful network connection is made, the process forks, and an *authentication context A* is created. This context is used for the lifetime of the login and is used to track a number of authentication based data values.

During the next step Key Exchange occurs, where the actual negotiation for a cipher, MAC and compression take place. First server authentication takes place via server/host key pairs. This authentication is transparent to the user if they have visited that SSHD server in the past. Assuming the server authentication is successful, algorithm negotiation for cipher and MAC takes place. Finally the short-lived session key is generated which is used to provide symmetric encryption for the data stream. This key is periodically re-negotiated after a given time or data volume passes. Since this is a reasonably well studied and logged area of

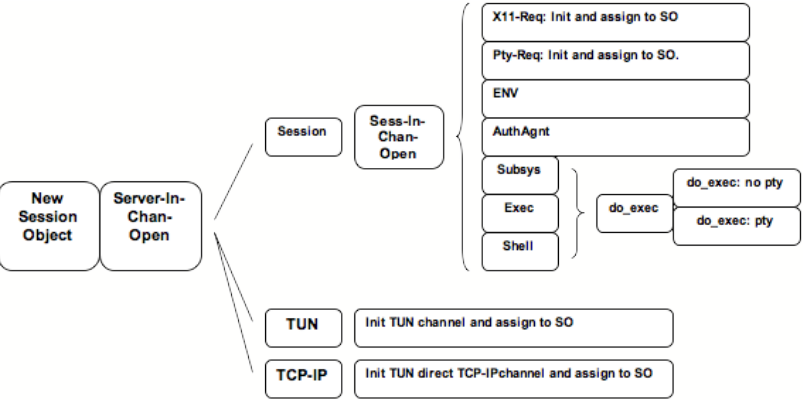


Figure 2: Internal SSHD Data Flow

the application, none of the exchange is recorded in the iSSHD besides what the system logs already do. If a strong reason to log the session crypto data could be come up with, there is no reason why it could not be done.

The Authentication Layer (unsurprisingly) provides the actual user authentication process. This process is extremely flexible with a number of options natively defined by the application as well as any generic PAM infrastructure. During the authentication process more than one type of authentication type can be examined so multiple fail and postpone events can be generated even for a successful login. Since we are less interested in the details of the authentication process than the outcome, there is little or no detailed logging from iSSHD except for the success/failure declaration as well as the authentication type being used. We apply the same rational to the key exchange process since in both cases relevant data can be preserved in regular system logs.

If the authentication process proves successful, a Session Object is created. This will be the primary container for not only the authentication context, but tty, X11 and channel data as well. The Session layer code also controls the mechanics of user login such as the login process, remote command execution, pty allocation and X11 forwarding.

The session object can create, use and destroy Channels. A channel can be thought of as a connection within the Session Object that has well defined semantics for data movement, windowing information, file descriptors and multiplexing capacity. Typically for a shell, you would allocate a single channel that holds the file descriptors for stdin, stdout and stderr. It is not unusual though to have many additional channels in use for X-windows, SOCKS forwarding and authentication agents. Data within a channel is not encrypted since it is contained within a session which already is. This is a critical point for monitoring which we will use to our advantage.

3.2 Common Code Paths During Execution

Now that the behavior of OpenSSH for a typical login has been described, we can more closely examine code paths for strategic places to insert instrumentation. Identifying those paths involved reading the source code as well as experimenting with sessions running in debug mode. Since the most common service for SSH to provide is remote shell login access, it was the initial target for both analysis and instrumentation. The execution path for this is identical to that shown in Figure 1, except for some additional details found in the session section. A location is considered a good candidate for auditing if (1) there exists a decision making branch where most or all connections traverse or (2) a final state is arrived at which contains security relevant information.

Figure 2 provides a more detailed set of code paths for nearly any use of OpenSSH. Here every box represents a transition between user privilege or application function and ultimately represents an event sent to the iSSHD analyzer. The creation of the Session Object (SO) begins on the left side and the path moves to the right till the users objective is reached. In it, a number of common paths that immediately stand out. The horizontal split between session and tunnel driven services is an obvious candidate for instrumentation. As a reminder, the session code tends to be more execution oriented – i.e. involved with the invocation of services, commands and shells. Since it is not unusual for an attacker to use a known tool or service in a way which is unusual, *how* we instrument the path is extremely important. Decision branches such as “session-in-channel-open” provide the path of what was asked for, and logging details at the end of the code path provide information regarding what was actually done. In any case, policy can be written to provide notice if the local site finds any part of the execution path objectionable.

Using the same rational, the lower half of Figure 2 provides the same opportunity to audit this behavior in some detail for tunneling and port forwarding activity. While not implemented in this design, it should be at least possible (though perhaps not practical) to access the forwarded data instead of just identifying the static forwarding requests.

The level of logging may seem excessive, but such detail can prove to be quite powerful for forensic analysis when combined with local site policy. Local site policy - described later in some detail - can act on specific session events like tunneling which may not be allowed by a centers usage policy. There is a huge benefit to be had in identifying the exact execution path of an attacker. Since it is not unusual for a tool like ssh to be used in a way which was not foreseen by the security community we tend to error on the side of caution.

4. SYSTEM ARCHITECTURE

For the iSSHD architecture, we selected three principles fundamental to the design and implementation process. If at any time one of these principles was in contradiction with the design, something was wrong with the architecture. The principles are:

1. **Avoid introducing stability or security problems**: We need to demonstrate with high confidence that our modified version of SSH is just as stable and secure as the original code base.
2. **Unchanged user experience**: The modified version of SSH can not affect the way users interact with NERSC systems, require a special version of the SSH client or application, nor remove any existing capabilities.
3. **Minimal impact on system resources**: System resources including CPU time, memory, and network bandwidth are at a premium. Additional demands made by the instrumented SSH must be insignificant compared to an unmodified SSH instance.

Based on these requirements, the following choices were made in the architecture and development plan:

1. **Use OpenSSH as the code base**. OpenSSH has an exceptionally good reputation and is already used on the multi-user production systems. In addition, we were able to add on the Pittsburgh Supercomputing Center's high performance OpenSSH patch set [12]. This provides significant gains in terms of bulk data transfer performance.
2. **Minimizing changes to the code base**. As part of the project we made an active attempt to minimize the number of changes to the original code. In addition, we chose to use other tools and capabilities rather than write them ourselves. An example of this would be the use of stunnel [20] rather than attempting to write an add on to ssh for our own data encryption.
3. **Decoupled Analysis**: Taking our experience from the Bro IDS, we chose to fully decouple the analysis from the generation of the ssh instrumentation data. To do this it was necessary to remove any dependencies between the running iSSHD and the back end analysis. This is done by making all writes to the back end non-blocking stressing that a failure of the analysis infrastructure should result in the loss of security data before an interrupted user experience.

The overall design of the iSSHD can be broken out into two sections – the event generation within the running iSSHD process, and the logging and analysis that compares those events against local policy. Much of §3 was involved with the thought process that took place before the coding started. With that in mind, we turn to the actual design and implementation of the system itself.

It should be noted that the core of the analysis side currently exists as a log repository with scripts feeding live data to the Bro IDS. The use of Bro is not technically required since the file exists as structured text, which provides the ability to feed the information to any another tool. We will assume for the remainder of the paper that Bro will be used.

4.1 Server Side

The iSSHD server is modified OpenSSH code that provides events for further logging and analysis. Within the SSHD application (as described in §3.2) there are ideal locations where we extract information about user activity. Such information includes login and authentication data, session and channel creation, port forwarding, and keystroke/application data. This data is normalized in terms of data types as well as being formed into structured text. This text is then written to a local socket (provided by stunnel) using a non-blocking descriptor. Details of this process follow.

For events, a number of data types are defined. Not unexpectedly these types map approximately with the native data types defined by Bro. This includes the usual integer, string and count as well as more network specific types like address and subnet. In order to encapsulate arbitrary data, both unstructured string and binary data is URL encoded using the stringcoders library [8]. This mechanism is used in reproducing user activity since even simple terminal sessions include Unicode characters and colors. An additional benefit of URL encoding is to safely encapsulate traffic that might be directed toward either the analysis system or the terminal session of the individual doing the analysis. Original versions of the instrumentation attempted to remove non-printing characters from the recorded data, but information loss and textual confusion ultimately pointed toward the URL encoding solution as a better option.

As has been already described, the most basic unit of information provided by iSSHD is called an *event*. Events, as their name implies, are single actions or decisions made by a user that are agnostic from a security analysis perspective. Lines typed by the user as well as logins and channel creations are all examples of events.

For event creation, all activity points to a single function. This reduces confusion and creates a single point for information gathering. A sample function call looks something like:

```
s_audit("channel_new", "count=%d count=%i
         uristring=%s", found, type, t1buf);
```

The function s_audit is the general event handling operation within iSSHD. There are three sets of arguments that it takes – the first is just the event name (in this case "channel_new"). The second defines data typing for the Broccoli interpreter and has printf() type structure. Any additional arguments define the data associated with the event type. Here, 'found' is the index for the free channel slot, 'type' defines the type/state of the channel (ie: SSH_CHANNEL_LARVAL, SSH_CHANNEL_AUTH_SOCKET), and 't1buf' is the URL encoded channel name such as *server-session* or *auth socket*. After passing through the Broccoli interpreter, an event named "channel_new" will be created with three arguments. Note that there is no indication that the channel creation is considered a good or bad thing – such a determination will be left to the analysis side of the iSSHD.

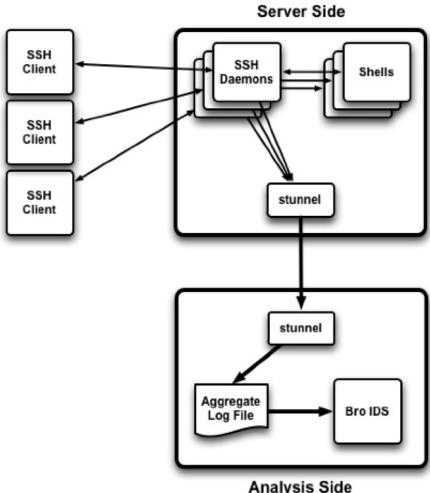


Figure 3: Overall iSSHD Architecture

Data provided by keystroke logging presents an interesting problem in that the content can be of arbitrary length, and will contain non-printing ASCII characters. To avoid inefficiencies, we cache keystroke data in a channel buffer queue using the native channel buffer types until a new line character is seen or data volume is exceeds a threshold. In situations where too much data is generated on the server side (such as large compile runs), the value of this additional data is almost zero. To address this, we adopted the same idea as used in the network Time Machine [7]: specifically that most security sensitive data and events tend to cluster them selves to the beginning of interactive sessions. By making the distinction between interactive sessions (where there are roughly the same order of magnitude of client initiated data events as server) and highly asymmetric connections (with dozens or hundreds of server data events per client data event), we can avoid excess resource consumption by the iSSHD. This is one situation where it was necessary to build logic into the code running in the iSSHD. Table 1 provides cutoff values for both normal tty channels as well as channels not bound to a tty. For the situation of non-tty communications, the ratio of printing to non-printing characters is also looked at to avoid needlessly copying binary files.

Table 1: Default cutoff values for user and server data.

TTY	Details	Default Value
Yes	Max line length or line count for client input between server inputs.	15 lines, 64k bytes
Yes	Max line length or line count for server input between client inputs.	15 lines, 64k bytes
No	Initial sample value (ISV) before determining binary data.	1024 bytes
No	Maximum data in total for either client or server inputs.	.5M bytes
No	Percentage of non ascii-printing characters, after ISV, allowed for continued sampling.	30%

For example if a user (client side) types 'ls -l' in a normal tty based login, the iSSHD would provide the server echo of 'ls -l' as well as the next 14 lines or 64k bytes of server side output (whichever is exceeded first). The line/byte count is reset every time client data is processed. The cutoff values are modifiable at compile time and are set somewhat conservatively since the assumption is that there is a large number of iSSHDs feeding into a single analysis system.

4.2 Data Analysis

Data analysis consists of any component except for the iSSHD itself. Practically it can be thought of as the stunnel as well as the bro instance and related policy.

The stunnel is not particularly interesting in that we are using it to transport data from an open file descriptor on the iSSHD side, to the analyzer host. Since this is just a simple implementation of a well-known application, we will focus on the details provided by the policy.

The bro policy is designed to track individual sessions and whatever activity is contained within them - normal shell sessions, remote code execution or subsystem invocation. Each session is defined by the start of the ssh connection and continues through any activity until that connection ends. The series of events for a routine login looks something like Figure 4 when printed directly from the iSSHD.

Each of these lines represents an event and the data associated with it. Policy can be written to trigger on specific events, their data, or both. Of obvious interest is a users keystroke data and the systems response. Since we have direct access to near real time keystroke information, we look for extremely unlikely - and highly suspicious - character sequences. These might include known toolkit signatures, abnormal root shell prompts for /bin/sh, or any other unexpected commands. Sets of commands that individually do not represent a significant interest, but which are suspicious in total represent the second type of alarm. These two categories are defined by two sets of signatures – the first for commands or strings worthy of immediate notification, and the second for sets of these commands or strings present in the user session.

In order to circumvent logging from the system login() facility, it is not unusual for attackers to remotely execute a shell via 'ssh host sh -i' . This style of reconnaissance has become so common during hostile activity that we made sure that it could be simply alarmed and all interactive data recorded. To address this, traffic on non-tty channels had to be tracked and analyzed since the tty invocation is part of the standard unix login() facility. Since data on these channels can include binary streams, the ratio of ASCII to non-ASCII packets is monitored. If after a pre-defined sampling window this ratio exceeds a threshold, further monitoring on that channel is dropped. We have experienced tremendous success in logging both the remote execution of shell binaries as well as monitoring commands to and from such occurrences.

```
SSHD_CONNECTION_START

AUTH_KEY_FINGERPRINT uristring=0x.. uristring=DSA
AUTH_INFO uristring=Accepted uristring=scottc
uristring=publickey

SESSION_NEW uristring=SSH2
CHANNEL_NEW  count=0 count=SSH_CHANNEL_LARVAL
uristring=server-session
SERVER_INPUT_CHANNEL_OPEN uristring=session
CHANNEL_NEW count=1 count=SSH_CHANNEL_AUTH_SOCKET
         uristring=auth+socket
SESSION_INPUT_CHANNEL_REQ count=0
              uristring=auth-agent-req@openssh.com
SESSION_INPUT_CHANNEL_REQ count=0 uristring=pty-req
SESSION_INPUT_CHANNEL_REQ count=0 uristring=shell

CHANNEL_DATA_SERVER count=0
 uristring=%0ALast+login:+Sat+Jan++8+14:45:31+2011
CHANNEL_DATA_CLIENT count=0 uristring=exit
CHANNEL_DATA_SERVER count=0 uristring=exit
CHANNEL_DATA_SERVER count=0 uristring=%0Alogout

SESSION_EXIT count=0 count=28221 count=0
CHANNEL_FREE count=0 uristring=server-session
CHANNEL_FREE count=1 uristring=auth+socket

SSHD_CONNECTION_END
```

Figure 4: Event series for a shell login.

The final area to explicitly mention is the ability of iSSHD to intercept authentication data. When considering our options for recording passwords during authentication, we ended up having to carefully balance the utility and risk of retaining the data. In the context of a forensic analysis, a password might be tremendously valuable if used in a legally sanctioned criminal investigation. On the other hand having such valuable credential information in the logs represents a huge risk in and of itself, even without taking into consideration passwords recorded for other institutions by users transiting local systems. Ultimately the decision to record passwords is left to the local site as a configure time option so that it cannot be adjusted without recompiling the iSSHD. Since it is not unusual for sites to share lists of known compromised keys via their fingerprints, public keys presented for authentication can be compared to a list of known bad keys and alarms raised when a suspicious key is seen.

4.3 Event Details

As previously suggested, events generated by the iSSHD are without any sort of predefined notions of good or bad since it is the role of the analyzer to interpret these events. These events can be roughly grouped by function, with types auth, channel, session, server and sshd. In addition to these, the sftp subsystem also has a number of events associated with it.

The example presented in Figure 4 shows the series of events seen in a "normal" login. Two of the most important in terms of monitoring and analysis are CHANNEL_DATA_CLIENT and CHANNEL_DATA_SERVER. These events provide unfiltered client keystroke and server echo/response data. If a user types "lz<backspace>s<enter>" you would see "lz%7Fs" from the client side and "lz%08+%08s" from the server side in the URI encoded data. The characters '%7F' and '%08' are the control characters delete and backspace respectively which can be seen from standard ascii definitions. Since we assume all user-generated data is potentially hostile, we reduce the possibility of accidentally interpreting control characters in the process of reading and interpreting the data by storing it in an encoded form.

Each event also includes timestamp, server id (process ID + server hostname + listening port), client id (32 bit random number) and interface address list. This information is tracked by the analyzer bro policy as a locally unique session identifier - for example #12345. This session id will remain constant for any activity attached to that users session. This event data is missing from figure 4 (and the other session figures) to allow for better clarity. Additionally, data is maintained for the channel id so session #12 might contain channel 0 and channel 1. Since the session object holds channel objects, the session id (ex #12) is the same and the channel identifier will be different. A small number of events, mostly connected to the running sshd daemon itself, do not have all these fields since there is no notion of client session to be had when the daemon is starting or emitting a heartbeat event.

5. RESULTS AND PERFORMANCE DATA

Presenting quantifiable results for the iSSHD is somewhat complicated since there is no control data to base comparisons against. Since the number of incidents is not large, checked against a control group or varied across sites, it presents more of an anecdotal story than an effective hypothesis test. Using iSSHD we have identified approximately three-dozen instances of stolen credentials. Most of them are not particularly interesting, but at the same time we can catch this class of attacker *before* anything can get interesting. Because of this, we will present an unusually qualitative analysis for the security and policy enforcement capabilities. For performance data we will look at a number of measurements comparing iSSHD to an unmodified version running on the same hardware. In addition we will also provide a simple analysis of aggregate user events that would be extremely difficult (or impossible) without the data set.

Besides detection, the iSSHD provides considerable insight into the tactics, skill levels and motivations for many of the attackers on our systems. In many cases the forensic logs quickly provide a clear indication of the success, skill level and threat presented by an intruder.

5.1 Sample 1: Remote Shell Invocation

Figure 5 provides a textbook example of a "classic" stolen credential and local exploit attack. This user (resu) made the mistake of having the same password for at least two sites - NERSC and the remote site that was compromised. Here the attacker remotely executes a shell to log in, then attempts a local linux exploit. Note that because of the shell invocation, communications are not via the normal tty interface - a technique detailed in §4.2 .

Details follow with some of the data fields removed for clarity.

```
1  AUTH_OK resu keyboard-interactive/pam
   1.1.1.1:52073/tcp > 0.0.0.0:22/tcp
2  NEW_SESSION SSH2
3  NEW_CHANNEL_SESSION exec
4  SESSION_REMOTE_DO_EXEC sh -i
5  SESSION_REMOTE_EXEC_NO_PTY sh -i
6  NOTTY_DATA_CLIENT uname -a
7  NOTTY_DATA_SERVER Linux comp05 2.6.18-…GNU/Linux
8  NOTTY_DATA_CLIENT unset HISTFILE
9  NOTTY_DATA_CLIENT cd /dev/shm
10 NOTTY_DATA_CLIENT mkdir ...
11 NOTTY_DATA_CLIENT cd ...
```

12	**NOTTY_DATA_CLIENT** wget http://host.example.com:23/ab.c
13	**NOTTY_DATA_CLIENT** gcc ab.c -o ab -m32
14	**NOTTY_DATA_CLIENT** ./ab
15	**NOTTY_DATA_SERVER** [32mAc1dB1tCh3z [0mVS Linux kernel 2.6 kernel 0d4y
16	**NOTTY_DATA_SERVER** $$$ Kallsyms +r
17	**NOTTY_DATA_SERVER** $$$ K3rn3l r3l3as3: 2.6.18-194.11.3.el5n-perf
18	**NOTTY_DATA_SERVER** ??? Trying the F0PPPPppppp__m3th34d
19	**NOTTY_DATA_SERVER** $$$ L00k1ng f0r kn0wn t4rg3tz..
20	**NOTTY_DATA_SERVER** $$$ c0mput3r 1z aqu1r1ng n3w t4rg3t...
21	**NOTTY_DATA_SERVER** !!! u4bl3 t0 f1nd t4rg3t!? W3'll s33 ab0ut th4t!
22	**NOTTY_DATA_CLIENT** rm -rf ab ab.c
23	**NOTTY_DATA_CLIENT** kill -9 $$
24	**SSH_CONNECTION_END** 1.1.1.1:52073/tcp > 0.0.0.0:22/tcp

Figure 5: Remote shell invocation example.

We can see a number of clear indicators that something is going on which is not normal user activity. First is the interactive session on a non-tty channel created by remotely executing a shell (line 3-5). Second, the unset HISTFILE command and the creation of a directory called "..." under /dev/shm (line 8-10). Finally the exploit is downloaded, compiled and (unsuccessfully) run (line 12-21). Highlighted text represents commands and output that as part of the default policy distribution are considered sufficiently unusual or dangerous to warrant alarming on.

5.2 Sample 2: Cluster Reconnaissance

This example is one of the more complex and educational that we have captured, providing a clear snapshot of the methodology and tactics taken by a pair of hackers looking into our systems. Since they are sharing a common login via the GNU screen utility we can see the interaction between them and get an understanding of their *methods and communication,* something quite difficult under normal conditions. While there are several thousand lines of interaction from the event, space limitations force us to only include a small chunk of the most interesting (and amusing) lines.

1	**DATA_CLIENT** /sbin/arp -a
2	**DATA_SERVER** b@n:~> /sbin/arp -a
3	**DATA_SERVER** comp05 (192.168.49.94) at 00:00:30:FB:00:00 [ether] PERM on ss
4	**DATA_SERVER** b@n:~>
5	**DATA_CLIENT** *oh wow*
6	**DATA_SERVER** b@n:~> *oh wow*
7	**DATA_SERVER** b@n:~> /sbin/arp -an |wc -l
8	**DATA_SERVER** 9787
9	**DATA_CLIENT** *rofl hax it hacker*
10	**DATA_SERVER** b@n:/u0> *sorry, im gonna s roll a cigarette and smoke it, y back*
11	**DATA_SERVER** b@n:/u0> *then im gonna come back and try to hack ok ?*
12	**DATA_SERVER** b@n:/u0> *i am gonna go for one*
13	**DATA_SERVER** b@n:/u0> *you cant smoke inside? terrible*
14	**DATA_SERVER** b@n:/u0> *its f cold as f****

Figure 6a: Initial communication and Note: removal additional server fields, time and session id

The text from the screen session is marked in blue, and event names are once again bolded. The overall behavior can be broken out into several sections. In Figure 6a, lines 1-10, arp tables are used to identify locally attached systems. In this case the large number of them (9787) seems to cause the need for a few moments thinking about how to proceed. This is one of the initial indicators that the attackers are not just blindly running tools. It also indicates that they are probably in the western hemisphere.

1	**DATA_CLIENT** hmm cd .. ;ssh-keygen -t
2	**DATA_SERVER** b@n:~/.ssh> *hmm*
3	**DATA_SERVER** b@n:~/.ssh> cd ..
4	**DATA_SERVER** b@n:~/.ssh> ssh-keygen -t dsa
5	**DATA_SERVER** Gen pub/private dsa key pair. ...
6	**DATA_CLIENT** ls
7	**DATA_SERVER** b@n:~/.ssh> ls
8	**DATA_SERVER** id_dsa id_dsa.pub known_hosts
9	**DATA_CLIENT** cat id_dsa.pub > authorized_keys
10	**DATA_SERVER** b@n:~/.ssh> cat id_dsa.pub > authorized_keys
11	**DATA_CLIENT** ssh -oHashKnownHosts=yes 192.168.0.1
12	**DATA_SERVER** b@n:~/.ssh> ssh -oHashKnownHosts=yes 192.168.0.1
13	**DATA_CLIENT** cat > ssh_cn010onf
14	**DATA_SERVER** b@n:~/.ssh> cat > ssh_config
15	**DATA_CLIENT** cat known_hosts | grep -v 192.168.0.1
16	**DATA_SERVER** b@n:~/.ssh> cat known_hosts | grep -v 192.168.0.1 > tmp ...
17	**DATA_SERVER** b@n:/tmp> *what are you trying to do get ride of t pressing yes?*
18	**DATA_SERVER** b@n:/tmp> *clearly*
19	**DATA_SERVER** b@n:/tmp> *lol set known_hosts to dev null n00b*
20	**DATA_SERVER** b@n:/tmp> *that is such a hack and completely improper*
21	**DATA_SERVER** b@n:/tmp> *and a good way to lose a box if you forget to remove it*
22	**DATA_SERVER** b@n:/tmp> *nononosec phrack.org done? wn? its in issue 64*

Figure 6b: Generate local key pair and populate across NFS share, attempt generic NFS type attacks via suid 0 program.

1	**DATA_CLIENT** ps axuw |grep snort
2	**DATA_SERVER** ps axuw |grep snort
3	**DATA_SERVER** b 36684 0.0 0.0 2740 564 pts/10 S+ 20:39 0:00 grep snort

Figure 6c: Looking for IDS processes.

By Figure 6b discussion has indicated a familiarity with insecure multi-host NFS file systems - interestingly, they did not attempt to use NFSShell. From here (lines 4-5) the pair generate a pass-phraseless ssh key to use across the systems sharing the home file system, once again indicating a familiarity with shared file systems and how they can be used. They grapple a bit with configuration issues and interestingly use the HashKnownHosts option to obscure records left in the known_hosts file. Figure 6c provides an example of IDS detection.

Ultimately this pair logged in to 19 local systems and never managed to get root access. The dialog here is as long as it is in order to convey the relative sophistication and interesting method of the attackers.

5.3 Performance Data

There are numerous points of reference in comparing the performance of the iSSHD with an unmodified OpenSSH. In this case we will be looking at aggregate remote command execution time, time to copy binary and ascii files, cpu usage for general activity, and memory usage for the child process.

This command set is run remotely via remote execution with the system time command providing information about total execution time, system and user cpu usage. We recognize the differences between remotely executing a script containing commands and manually running them. Ultimately we chose to run via the script for repeatability and ease of use since tools such as Expect do not provide additional functionality.

	Remote Exec	SCP Binary	SCP ASCII
SSHD	42.78 [0.05]	9.85 [0.11]	0.70 [0.01]
iSSHD	43.03 [0.18]	9.85 [0.15]	0.69 [0.02]

Table 2: Run time values for three tests, values in seconds, standard deviation in brackets. Average remote command execution time increases by 0.6%.

For Table 2 column 1, "Remote Exec" is a set of 13 remotely executed commands including normal user activity like ls, touch configure and make. From a simple ratio test, the iSSHD takes in total about 0.25 seconds more to run or about 0.6%. This indicates that the *average* behavior of interactive shell commands should not be adversely affected, but limited variations in keystroke responsiveness could be lost. Given the way that large volume logging is done (as described in §4.1), this is not at all surprising. For the additional columns in Table 2, we have the time to completion values for using scp to transfer a medium size ASCII file as well as a medium size binary file. In this case, medium size is on the order of 100MB. In each case the additional overhead caused by the memory copy and transmit did not provide a significant (or measurable) difference in the measured time. In this measurement, the same file was moved from one directory on the local system to another 40 times in a row. The task was then repeated with the iSSHD to reduce the influence of variable overhead and caching.

Looking at CPU usage for the same two data sets demonstrates differences in application behavior. First, the system CPU dominated the total time by ~ 4:1 for total CPU time per transaction. This is not surprising given that the majority of this activity is driven by read() and write() calls as well as polling during periods of inactivity.

Figure 7 shows the relationship between execution time and CPU time for both sets of test runs. One thing to notice is the slope of the linear regression curve. Total CPU usage decreases since the faster you move a constant set of data, the harder the data must be pushed during the (shorter) time window. The product of the two terms as a histogram we see a very tight set of values (s^2), implying this relationship.

The final metric is memory use, which ends up being quite consistent both in terms of native and iSSHD when looking at results from the data generation scripts. Within SSHD, there are a limited number of ways that memory becomes allocated once a session completes initialization – the most common being internal data buffering and channel creation. In both of these cases the size growth is minimal for the modifications made since data buffering from interactive sessions are cleared once they are written to the stunnel socket.

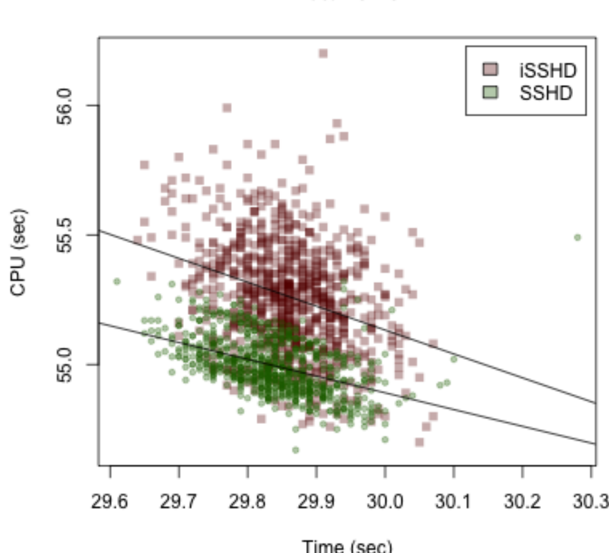


Figure 7: Total CPU time vs. length of transaction time for test data runs against iSSHD and native SSHD.

The overall conclusion is that the changes made to introduce instrumentation into iSSHD do not have a significant impact on performance or usability.

5.4 Overall Observations

Overall the iSSHD project has provided insight into probably three-dozen compromised user accounts since 2009. In each of these cases it was possible to not only quickly determine the success of the attack, but also get exploit tools and code used.

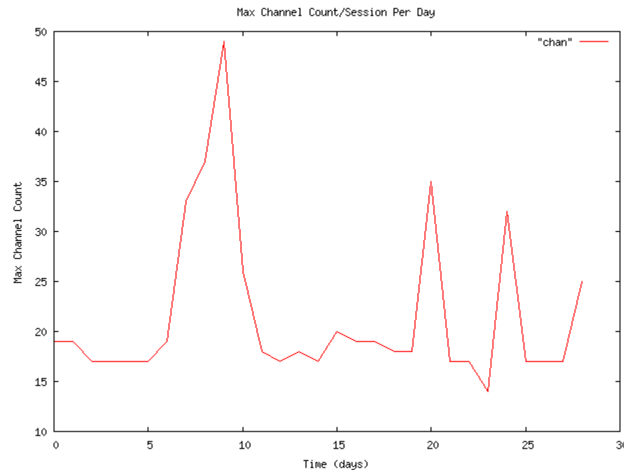


Figure 8: Distribution of maximum channels/session for November 2010.

As suggested in the introduction, the iSSHD also provides a tremendous source of measurement data as well. We have not yet

begun to fully explore this avenue, but there is no technical reason why we could not use this to identify needs for the user community. An example of this would be to systematically explore port-forwarding behaviors to see if we could deliver network services differently. Besides problem solving, the measurement data can also provide an interesting repository of pure research data. Figure 8 provides an example of the maximum channel count per session per day during November 2010). It is interesting to note that some users are exceeding 50 channels per session – in this case the majority of this is web browsing. This might be done (for example) to visit social networking sites blacklisted by a users local institution. This has interesting security repercussions to be sure.

6. FUTURE WORK

Since the iSSHD is relatively new, there is a great deal of learning going on with regard to what information is useful as well as available. There are several areas that we are actively looking into for future releases. The first is the detection of local terminal session hijacking as described in §2 by [17][18]. The second is the extraction of keystroke data from the X11 x-terminal data-stream, which is currently opaque. There is currently some prototype work completed for the session hijacking (detailed below), while tapping into the X11 stream represents a possible way to look into the protocols being tunneled over the ssh channel.

6.1 Local Session Hijacking

In the available literature and toolkits, there are a number of ways that a local attacker can tap into a running session and "reach across" the network to access further systems and resources. In particular this can be done to elevate privilege if the user has gained root access on the external system, or to hop over one time password authentication. We are familiar with examples of the later.

In the SSH-Jack application [17], ptrace is attached to the ssh client process, finds the channel setup code, then patches the memory to request a remote shell attached to a local TCP socket. The user running the ssh client is completely unaware that this is happening since they are running under a different set of channels in the same user session. We are hoping to look for an unusual ssh_session2_open() call and match it to the expected state for a normal session to help identify this attack. Regardless of this, the entire communications from the new channel will be logged and analyzed in the same way that normal user activity is.

A more common attack involves a local root user looking to jump off the compromised host through some sort of multi-factor authentication. In many cases this involves the opening of the victim users terminal descriptors for standard in, out and error then writing data directly into the sockets. The running ssh is not even aware that anything is amiss since it is just transiting data normally. We are looking to use the Linux *inotify* interface [2] to monitor and log additional file open events on the terminals file descriptors. This is still in its prototype phase.

7. CONCLUSION

We have presented an instrumented version of the OpenSSH application that allows for a local site to log and analyze user activities on local HPC resources. This analysis can be used to enforce local security policy with respect to SSH usage, which would otherwise be difficult or impossible with normal tools.

8. ACKNOWLEDGEMENTS

This work was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

I would very much like to thank Tom Limoncelli for his help in the paper shepherding process.

9. REFERENCES

[1] "Trust Transience: Post Intrusion SSH Hijacking" to Blackhat Las Vegas, Adam Boileau

[2] Dow, Eli M., *Monitor Linux file system events with inotify*, IBM Linux Test and Integration Center, http://www-28.ibm.com/developerworks/linux/library/l-inotify.html?ca=dgr-lnxw07Inotify, 2005.

[3] H. Dreger, C. Kreibich, V. Paxson and R. Sommer, Enhancing the Accuracy of Network-based Intrusion Detection with Host-based Context, Proc. Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) 2005.

[4] C. Kreibich and R. Sommer. Policy-controlled Event Management for Distributed Intrusion Detection. 4th International Workshop on Distributed Event-Based Systems (DEBS'05), 2005, Columbus/Ohio, USA

[5] Joohan Lee, Muazzam Siddiqui, "High Performance Data Mining for Network Intrusion Detection Using Cluster Computing", International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), MIT Cambridge, November 2004

[6] Malek Ben Salem, Shlomo Hershkop, Salvatore J. Stolfo. "A Survey of Insider Attack Detection Research" in Insider Attack and Cyber Security: Beyond the Hacker, Springer, 2008

[7] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson and F. Schneider, Enriching Network Security Analysis with Time Travel, Proc. ACM SIGCOMM, August 2008.

[8] Nick Galbreath, stringencoders: A collection of high performance c-string transformations, http://code.google.com/p/stringencoders/

[9] Open SSH Project, http://www.openssh.org

[10] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time. Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998

[11] http://packetstormsecurity.org/files/view/42556/phalanx-b6.tar.bz2

[12] Chris Rapier and Benjamin Bennett. 2008. High speed bulk data transfer using the SSH protocol. In *Proceedings of the 15th ACM Mardi Gras conference,* (MG '08). ACM, New York, NY, USA, , Article 11 , 7 pages. DOI=10.1145/1341811.1341824 http://doi.acm.org/10.1145/1341811.1341824

[13] Know Your Enemy: Sebek. The Honeynet Project, November 2003 https://projects.honeynet.org/sebek

[14] SSHD Backdoor Homepage, http://emsi.it.pl/ssh/

[15] http://packetstormsecurity.org/files/author/5480/ : Backdoored version of OpenSSH 4.5p1 that logs passwords to /var/tmp/sshbug.txt.

[16] http://packetstormsecurity.org/files/view/45228/ssheater-1.1.tar.gz : SSHeater is a program that infects the OpenSSH daemon in run-time in order to log all future sessions and implement a backdoor where a single password, chosen by the user, can log into all accounts in the system. There's a log parser included in the package that can display authentication information about sessions as well as play the session just like TTYrec/play.

[17] http://www.storm.net.nz/projects/7

[18] http://datenterrorist.wordpress.com/2007/07/06/tty-sniffer-fur-linux-24/

[19] SSH CRC32 attack detection code contains remote integer overflow, Vulnerability Note VU#945216, United States Computer Emergency Readiness Team, http://www.kb.cert.org/vuls/id/945216

[20] W. Wong: Stunnel: SSLing Internet Services Easily. SANS Institute, November 2001.

[21] W. Yurcik, X. Meng, and N. Kiyanclar. NVisionCC: A visualization framework for high performance cluster security. In ACM Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC), 2004.

[22] W. Yurcik, Chao Liu, “A first step toward detecting SSH identity theft in HPC cluster environments: discriminating masqueraders based on command behavior”, CCGRID '05 Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01

[23] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Protocol Assigned Numbers”, RFC 4250, January 2006

[24] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Protocol Architecture”, RFC 4251, January 2006

[25] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Authentication Protocol”, RFC 4252, January 2006

[26] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Transport Layer Protocol”, RFC 4253, January 2006

[27] T. Ylonen, C. Lonvick, “The Secure Shell (SSH) Connection Protocol”, RFC 4254, January 2006

Appendix 1

This is an abbreviated list of iSSHD events current as of January 2011. The event name is in the left column and a summary of returned data types is on the right. All events are processed in the current public release of the policy set. The description in the Returned Data column does not include all the default fields described in §4.3.

Authentication Events	Returned Data
auth_info	userid, auth type, success, source IP, dest IP
auth_invalid_user	userid
auth_key_fingerprint	fingerprint of pub key
auth_pass_attempt	userid, password

Channel Events	Returned Data
channel_data_client	URI encoded client data
channel_data_server	URI encoded server response
channel_data_server_sum	Data skipped by heuristics
channel_free	id of closed channel
channel_new	id, type, remote name
channel_notty_analysis_disable	printable/non-printable ratio for non-tty channel exceeds set ratio
channel_notty_client_data	URI encoded non-tty client data
channel_notty_server_data	URI encoded non-tty server data
channel_pass_skip	id of channel where pass skip happened
channel_port_open	type, listening port, path/hostname, remote ip, remote port
channel_portfwd_req	hostname, listening port
	type, listening port, path/hostname, remote ip, remote port
channel_post_fwd_listener	listen port, path/hostname, host port, type
channel_set_fwd_listener	type, wildcard bind, host, port to connect, listen port
channel_socks4	id, path/hostname, host port, s4 command, username
channel_socks5	id, path/hostname, host port, s5 command

Session Events	Returned Data
server_input_channel_open	chan_type, channel, window size
session_do_auth	session type, state
session_exit	chanid, parent pid, status
session_in_channel_req	chanid, chan type, session id
session_remote_do_exec	parent pid, command
session_remote_exec_no_pty	parent pid, command
session_remote_exec_pty	parent pid, command
session_request_direct_tcpip	orig host, orig port, dest host, dest port, session id
session_tun_init	tun type, can id
session_x11fwd	display as string

SSHD Events	Returned Data
sshd_connection_end	remote ip, remore port, local ip, local port, client id
sshd_connection_start	remote ip, remote port, local ip, local port, parent pid
sshd_exit	local ip, local port
sshd_restart	local ip, local port
sshd_server_heartbeat	select value
sshd_start	local ip, local port

Adventures in (Small) Datacenter Migration

Jon Kuroda, Jeff Anderson-Lee, Albert Goto, Scott McNally
University of California, Berkeley, EECS
{jkuroda,jonah,goto,mcnally}@eecs.berkeley.edu

Abstract

In May 2011, we embarked on an ambitious course – in 3 weeks: clear out a small, soon to be demolished, research datacenter containing 5 dozen research systems spanning 5 research groups and, along with a new faculty member's systems located off-site, move it all into another space suffering from 18 years of accumulated computer systems research history. We made it happen, but only after intensive pre-planning and after overcoming a number of challenges, both technical and non-technical, and suffering a moderate amount of bodily injury.

We present an account of our adventures and examine our work in facilities, networking, and project management and the challenges we encountered along the way, many of which were not primarily technical in nature, and evaluate our approaches, methods, and results to extract useful lessons so that others may learn from our reckless ambition.

Introduction

In late 2010, the EECS Department revisited previously shelved plans to renovate the north half of the 5th floor of the Computer Science building, Soda Hall, including demolition of 600 ft^2 of datacenter space ("530") shared by 5 active research groups and 1 defunct group. Mercifully, the new plans left alone a network closet that had been originally slated for a relocation but proved far too costly to move.[1] The demolition schedule shifted a bit but, by mid-March, eventually settled down around late May/early June with a construction start date of 2 June announced in late April.

Concurrently, we had a new faculty member bringing a rack full of research systems from a nearby Industrial Research Lab in Downtown Berkeley ("IRB") that needed datacenter space and needed to move by 25 May. We also folded this additional, smaller, migration into the overall plan.

With a larger campus and departmental reexamination of the utilization of datacenter space, we also saw this as an excellent opportunity to clean up, reorganize, and plan to upgrade our remaining datacenter space.

Others have examined the topic of large datacenter-scale change, most specifically Cha who looked primarily at the computing side of migration, especially system configuration, while we tightly controlled the amount of system configuration change and were also heavily involved in the facilities/physical plant side of the migration.[2] Similarly, Cumberland focused exclusively on system installation and configuration while our systems were already up and running and could not be wiped arbitrarily.[4]

Location Location Location

In evaluating new locations for migrating systems, we considered 5 major characteristics:

- Air Conditioning (measured in tons[2])
- Power (measured in kVA[3])
- Space (measured in number of racks)
- Existing Network Access
- Ease of Moving Systems

We considered 6 locations for evaluation, but only one made sense for relocation – a 1000 ft^2 datacenter space on the fourth floor of Soda Hall ("420A"). It had several points in its favor, including the most surplus cooling capacity and, after a number of upgrades, power, some pre-investment in overhead fiber distribution systems[4], sufficient connectivity to relevant networks, and physical proximity to related research groups, staff, and 530. Other facilities were either already or about to be filled to capacity, poorly suited for experimental systems requiring frequent physical access, accessible by too large a group of people, or lacked sufficient network access.

Like all such facilities in the building, both used raised floors with underfloor forced air cooling sharing space with underfloor power distribution and legacy network fiber runs. They were similar in most ways with 420A being a larger version of 530.

Table 1 summarizes the nominal capacity and actual usage of 530, IRB, and the final destination, 420A. At first glance, these numbers seem to indicate that moving these systems into 420A would run very close to if not right up against the rated capacity on Cooling and Power, but table does not take into consideration the later removal of defunct systems and the occasional but generous rounding up of usage numbers by Facilities staff.

Facility		AC tons	kVA	#Racks
530	Rated	30 (2× 15)	50	15
	Used	10	25	5
IRB	Used	2-3	10	1
420A	Rated	30 (2× 15)	100	34
	Used	13	45	12

Table 1: Facilities Utilitization Summary

While the best (or least-worst) choice, 420A still had several points against it, all centered upon its age, including an 18 year-old raised floor that had never been cleaned, 18 years of accumulated cable tangle stemming from minimal management of underfloor power and legacy fiber distribution, a multitude of circuit types instead of a single standard, 18 years of systems research history (aka "junk"), and a disorderly mix of standalone cage racks and relay racks.

Though the Computer Room Air Conditioners (CRACs), essentially in-room chilled water heat exchangers installed in pairs in each room, were nominally up to the task of handling the additional load, they too were 18 years old and not running at maximum efficiency. In fact, the manufacturer sold off that division shortly after Soda Hall opened in Fall 1994, and we are no longer able to get manufacturer replacement parts such as logic control boards – many of these units have had custom replacement boards installed by a third party vendor. A mix of "ownership" issues (the campus physical plant, not the department, manages the building HVAC system including the CRACs) and budgeting issues (the availability of funding for operational expenses versus funding for capital expenses in physical plant's budget) complicate outright replacement.

Power distribution in both locations consisted of an in-room PDU taking 3-phase input feeding under-floor runs of both rigid metal and armored flex-conduit (referred to as "cable snakes" locally) carrying single phase circuits ranging from 15A to 30A and 120VAC to 208VAC. After 18 years of use by a succession of resource-hungry computer systems research endeavours (each with its own power requirements) with minimal efforts to manage the power distribution, "cable snake" became an increasingly accurate term as the underfloor area devolved into an increasingly difficult to manage tangle of flexible conduit with many circuits disconnected but left under the floor, interfering with orderly air flow and creating a maintenance nightmare for the current staff.

We could address these problems given enough effort, which we had; time, which we had in short supply; and funding, which we had but is complicated in our academic environment though eventually tractable given enough time. For our immediate needs, this facility badly needed a thorough cleaning, a complete electrical survey before making any needed changes, and another thorough cleaning – these would be our immediate priority while other concerns would have to be dealt with as longer-term projects.

Timing is Everything

In our academic environment, we try to schedule major work involving downtime after May for many reasons. Finals, class projects, post-semester research retreats, conference/journal submission deadlines, and even VLSI tapeout schedules can all key off of the end of the semester, but we can engage in major work with relative impunity in the brief 2-3 months between the spring and fall semesters. Unfortunately, this also applies to major construction work, so we had to choose which of March, April or May would be the least disruptive time to accomplish this feat.

We went with May, partly due to circumstance, partly by design. Availability of the department electrician and a trio of work-study student staff would prove crucial, but they were committed to other work until May. Looking for an upside to this, we found this gave us more time to prepare and plan so that, when May rolled around, we could spend more time working instead of backing out of costly on-the-spot decisions made with little forethought or waiting to work because we had not thought something through extensively enough.

This choice had obvious downsides. While the demolition schedule carried a 2 June deadline, 21 May proved much more relevant due to off-site research retreats and long-scheduled staff travel in the last two weeks of May which gave us 3 weeks of time with the entire team present to do the actual work of prepping the new space and moving systems while power shutdown of 530 was scheduled for 25 May to allow for dismantling and disposal of the two CRACs. This aggressive schedule came back to bite us once or twice, but the hard and very real deadline proved to be very strong motivation for us and gave us greater ability to cut through bureaucracy –

pushback from those outside our team or claims that we "could just get an extension" were rebuffed with a reminder that contractors were arriving on 2 June and that delay would hold up a major construction project, and, if necessary, an invitation to discuss the matter with the department chair. This, however, only happened once with a colleague who was unaware of the entire scope of the work and was quickly handled.

Our faculty, in particular, left us alone to do this work and did not question the migration schedule. While they already trust us in general on operational matters, they had specifically been informed of this work by the department chair and the facilities director to prevent exactly these sorts of questions – there was faculty buy-in to this adventure before it even became our concern. While we may not have particularly wanted to go down this road, at least this road had already been paved for us. Of the 2 other active groups who had systems in 530, one group would be moving into the renovated space and the other had only a small handful of systems and did not mind much as long as the systems were back up eventually – we did not anticipate nor receive any pushback from these two groups on the migration schedule. That we received no pushback on schedule from faculty or users and almost none from staff still amazes us.

People Get Work Done

One immediate challenge we faced was that we have no staff dedicated to Datacenter Management or who have it as a primary job responsibility. Instead, we have a number of staff who do work in datacenters as one aspect of their jobs, whether they be systems administrators, facilities managers, or network administrators. Most notably, the systems administrators responsible for completion of this project all work directly for 3 of the 5 research groups affected by this move – the other 2 research groups did not have systems support staff to contribute to the overall migration.

Our team consisted of one Department Facilities Director, one Department Electrician, three systems administrators with deep institutional knowledge, one new guy who started in May, and three work-study student helpers. Missing from our merry band was a network administrator to handle the myriad of network changes needed for all this – primarily a number of changes in VLAN assignments. The staff member most familiar with the network involved in this migration had taken another position elsewhere on campus in February 2011 and left behind another network administrator unfamiliar with both the overall topology as well as the platform specifics. This hole would come to haunt us later and nearly derailed the migration schedule.

Close ties developed over the past decade between team members proved vital to success given our tight timeline – having to go "through channels" for change requests and having to continually re-establish a shared terminology would have crippled our ability to move quickly on a tight schedule. We did, however, observe a marked discrepancy in the preferred manner of communication – while we all relied on 1-to-1 in-person communication for low-latency high-bandwidth communication, we never converged on a single mailing list, wiki, or any other particular form of asynchronous collaborative "hivemind". Periodic synchronous standup meetings proved to be only the consistent way to keep us all on the same page.

Goals

With limited staff and time, we had to keep our goals modest while at the same time ensure that we allowed for future facility improvements and upgrades. As noted, we had already decided not to engage in major upgrade or reorganization work in 420A. We also had to decide how much support to give to systems that did not belong to our faculty but instead belonged to the two groups without systems administrators.

In the end, we settled on providing a minimum baseline level of service of rackspace, power, and networking for all systems migrating out of 530 but only systems that belonged to our faculty received hands-on support from us. We convinced the Department IT Director to take responsibility for systems that belonged to a defunct research center whose sole faculty member had retired years earlier. Of the two groups lacking systems administration staff, one chose to hire the department's User Support Group to handle the hands-on migration work while the other gave the work to one of their undergraduate interns.

For our own faculty's systems, we settled on 3 service guarantees:

- Max of 1 downtime/system
- Max of 1 day/downtime
- Minimize impact on deadlines
  - do not move everything at once

We aimed to have systems back up within 24 hours after taking them down for migration and, once we said a system was back up, for it to stay up barring user needs or "normal" routine operational needs such as periodic OS patching. We particularly wished to avoid taking systems down again to move after announcing that a system had been moved and was back up.

The last goal proved to be the most complex but also the one most beneficial to us. We decided early on that moving everything all at once in one fell swoop was far

too disruptive to our users and risky for us if we took a wrong step, so we instead chose to move systems based on when relevant user populations needed them – moves more than 2 weeks before a deadline or the day after a deadline were acceptable, but not during the two weeks before a deadline. This would ultimately benefit us as we could pick and choose unused and therefore less critical systems to move first in order to test the waters after which we could move systems in larger groups.

For the new faculty member's IRB systems, we focused on:

- space for racked systems
- installation of electrical circuits
- transport on or before 25 May

We were not immediately concerned with getting the systems from IRB up and running, only with making sure that there was a location for them and making use of the electrician's time while we still had access to him in May.

When possible, we allowed our decisions to be guided by the pursuit of progress towards a cleaner, more well organized datacenter space, but were prepared to make well-defined and easily undone short-term decisions in order to meet our 21 May deadline.

Space Planning

We had actually begun planning for this over a year prior when the renovation plans first came across our desks. Already dissatisfied with the collection of ad hoc changes made to datacenter spaces throughout the building and the way that they hampered any growth or reorganization, we all saw this as an opportunity to rip out as much cruft, junk, and accumulated history as we could manage in whatever time frame we could acquire.

While we could not muster enough momentum or staff time to accomplish significant datacenter cleanup after the department shelved these initial plans, the ideas for cleanup and upgrade were still fresh in our minds and on paper when the department took the renovation plans back off the shelf. Additionally, we had already done a survey of 530 and 420A in late-2009 as part of a campus datacenter utilization survey, so we had a good handle on who had what systems in 530, how much power they consumed, and how much rackspace they needed.

Initial migration planning started with a overall survey of 530 to review any major changes since the 2009 survey. We paid special attention to the type of circuits we would need in 420A – while individual systems used standard IEC 60320 electrical connectors such as found on typical PC systems, our in-rack power distribution used a variety of means to connect to building power including 4 different NEMA twist-lock connectors. We conducted a similar general survey of 420A to confirm general impressions of the space and to note things that we could correct before May without the assistance of the electrician or the need for the trio of work-study students.

We briefly entertained the notion of installing an overhead busbar power rail system, as is now increasingly common in new facilities on campus, but quickly placed it on the "needs time and money" list. Within the time constraints we had, particularly the electrician's availability, we had to make do with more limited incremental changes to the existing underfloor power distribution system. Overhead power would become one of many recommendations that we would make for future datacenter upgrades.

We also held off on any major upgrade work to the air conditioning system, again for reasons of time. In place of capacity upgrades, we pursued two alternatives. First, the Facilities Director ran two day long experiments running 420A with 1 out of 2 CRACs shutdown to see if each of the 18 year old pieces of equipment could handle the existing load alone – which they did without failure. While not a strictly rigorous experiment, it demonstrated that we could have enough cooling capacity given prudent placement of systems and pruning of unused or offline systems. It also enabled us to pursue a second avenue – maintenance service and overhaul. The Facilities Director scheduled two maintenance periods for each CRAC involving aggressively proactive replacement of worn parts, cleaning of water piping to and from the building's rooftop chilled water supply, and servicing of each CRAC's 3 compressors. Our Facilities Director estimates that this restored about 20-30% of efficiency back to the CRACs though he notes that building AC makes exact numbers difficult to obtain – in warm months, building AC runs more often, creating a shell of cooler rooms surrounding 420A while in cooler months overall need for heating is rare due to the effectiveness of the building's own insulation.

We reviewed several ways to rearrange 420A, but ultimately retained the existing arrangement of 4 main rows of racks and 1 catch-all row with a more concerted effort at enforcing hot and cold aisle separation for dense installations while relegating less dense installations to "warm" aisles. We did rearrange rack allocations so that projects, which previously had equipment strewn across various disparate racks due to the floorspace equivalent of disk fragmentation, could benefit from physical proximity, thus alleviating network fiber distribution complexity as well as strengthening project and group identity. In addition, we identified equipment from prior projects which had been abandoned in place and was eligible for reuse or salvage.

Once we established an initial rack-by-rack layout,

mapping research groups to racks, we worked out which systems would go into which rack and ultimately came up with a rack-unit (RU) by rack-unit layout. While we could not move any systems until May and knew that plans could change in an instant, this gave us a start on planning in-rack power and network wiring and allowed us to organize systems in a more sensible fashion compared to the previous "which rack has room?" method.

After running through a few different ways of sorting systems into racks, we eventually identified 3 classes of systems that aligned naturally with their owners and purposes that also lent itself to a means to organize the racks and to answer the inevitable question of "where do I put this system?"

Experimental research systems
Our faculty's systems
Sat on "Research" network
Novel Hardware is the point
Often had two of each kind

"Production"ish research systems
Our faculty's systems
Sat on "Research" network
Stable research platforms/services
Clusters, OS dev, storage

"EECS" systems
(mostly) Other faculty's systems
Sat on "Department" Network
Group web servers, SW

Our eventual rack layout would later reflect these alignments and would let us make use of some limited "luxury" hardware resources more effectively. For instance, while we did not have the resources on hand to put a UPS in every rack, we did have a new-in-box UPS that had been bought but was never used for a now-decommissioned system – we installed that UPS into the Production rack to provide battery backup to a 24TB storage system and to some management systems. Similarly, other racks got "intelligent" power strips with remote outlet control and per-outlet power metering that, while not so useful for research quality results, let us keep tabs on spikes in power consumption as researchers ran experiments. Meanwhile, racks housing systems that we expected would see frequent hardware reconfiguration got an in-rack KVM so we could avoid having to un-rack systems just to get a console on them. Most of this hardware (cabinets, power-strips, UPSs, KVMs) was re-purposed from prior projects.

As the end of April approached, we began more definitive preparations. One admittedly sneakier one was to make daily sweeps of systems in 530 to check for running user processes and when the last non-staff login occured. If no user processes were running and the last non-staff login was more than a few weeks prior, we preemptively shut the system down. We shutdown 15 systems this way and only had to turn one back on – the remaining 14 remained powered down until we moved them at our relative leisure in May.

Some of the more obvious space preparations included basic cleanup of the space – we lost track of the number of cardboard boxes we had found squirreled away in every possible corner, nook, and cranny – collection and removal of abandoned or deprecated systems that had been left behind in racks but never tagged as excess, and tagging of equipment for storage and later repurposing. In preparation for the inevitable exodus of unused systems from both 530 and 420A that nobody was quite prepared to send to the campus "Excess and Salvage" unit, the Facilities Director had begun his own cleanup of a large basement storage room for our use during the move.

By mid-April, we had a good handle on the work needed to ready 420A – outside of a fair amount of hands-on physical labor, we saw no major facilities obstacles to having space ready for move-in during May, leaving only networking left as a major concern.

Network Planning

We had three areas of concerns regarding the network changes needed to support this move: the changes needed, who would do the work, and, as usual, the time available. As with planning for other parts of this move, we chose to stick with minimal changes instead of an overly ambitious redesign.

The systems moving out of 530 were spread across two distinct networks, a "Department" network, managed and funded centrally by the department, that provided general commodity network connectivity throughout the department and a "Research" network, funded by research grants and donations and managed by research systems support staff until early 2011, which evolved out of a wider network deployed for a campus-wide clustered computing project to serve the needs of specialized computer systems research in the Department. The vast majority of systems belonging to our faculty sat on the Research Network while the dozen or so systems that belonged to other faculty sat on the Department network.

This presented one small but immediate problem. While 420A historically supported systems associated with clustered and distributed computing research – the projects involved in such research provided their own network hardware to support the higher density networking they required – the Department network had very little presence in 420A at all, no more than a dozen ports available via network drops pulled in from a nearby network closet that were meant for one-off systems, not for higher density installations. Department network-

ing staff did not have any spare equipment to install a managed switch to support denser installation in 420A of systems on the Department network, but fortunately research networking had enough spare equipment to loan out a switch which someone would setup as a managed switch attached to the Department network. The last detail about it being used as a managed switch would change, but this plan would remain otherwise unchanged.

A larger question was what direction to take the Research Network's presence in 420A. The Research Network's presence in 420A, once extensive to support large clustered computing projects, had itself dwindled in size as systems' power and cooling density rose far faster than their space and networking density[5] and by this time had evolved into a few network stubs supporting smaller projects.

One such stub, a group of 4 daisy-chained switches attached to the Research Network via a lone 10Gb/s link over long-range fiber, was on the right VLANs which opened up the possibility of just daisy-chaining even more switches. We had previously discussed plans to stem the growth of the Daisy Chain of Doom (DCOD) but lacking sufficient spare long-range optical modules for a second long-range fiber run, extending the DCOD was straightforward, predictable, and cheap since we did have plenty of switches and short-range optical modules. We understood the downsides of relying on what would turn out to be a daisy-chain of 7 switches, but felt that it would be acceptable for the short-term (6-9 months) until we could spend enough time planning more extensive changes.

Regardless of any physical topology changes, we quickly realized that there would be a significant number of changes to port VLAN assignments to support systems with private interfaces on a separate VLAN. This led to the biggest question – who was going to do the work of reconfiguring the switches?

Prior to 2011, research systems support staff shared management access and duties with a lead "Network Guy" who himself had worked in our team as a split network and systems administrator before transitioning in 2009 to a full-time network mangement position supporting both the Department and Research Networks. Upon his departure, he handed over the Research network to the remaining network administrator at the direction of the department IT Director who wanted to see both networks managed in a more unified manner.

We were wary of this change, specifically losing access to manage the research network, but, in a good faith attempt to support the IT Director's direction in network management, we went along with his request that we send all of our network change requests to the department network staff which consisted of the remaining network administrator and the soon-to-retire infrastructure services manager. While we expected some communication and culture differences, we did not anticipate at this point the delays that would come with this and nearly derail the migration.

In late April, we met with the remaining network administrator to go over high level plans and examples of the changes we wanted so that everyone was on the same page. We held one more meeting to go into further detail and left with everyone understanding what we needed and when we needed it. At this point, every major task was identified and assigned to one or more persons.

Progress

Once May started, we gained access to the department electrician, work-study student labor, and, much to our delight, a large storage room courtesy of the Facilities Director for anything and everything we wanted to remove from 420A or 530 but were not quite ready to junk. Also joining us on 2 May was our new systems administrator who showed up for work right as we began the bulk of the work. We all met on 1 May for an initial meeting to confirm that we were all on the same page, and from that point, work progressed quickly.

Our first order of business was a detailed electrical survey of 420A so we actually had some idea of what circuits were actually live. At the same time, we were tagging equipment either to go to Excess and Salvage or to storage for the work-study crew to remove from 420A which they did as fast as we could tag it. Within a week, the electrician had mapped out all circuits in 420A, made all of the initial electrical changes that we had requested, and disconnected all hardwired powerstrips that had been attached to the legacy relay racks. We would later ask for a few changes which were quickly handled. Once this work was completed, the work-study crew set to work removing relay racks so we could bring in cage racks from storage that we had setup in a staging area with in-rack power distribution and cable management.

In the second week of May, we had begun to bring in networking to individual racks. While we waited for the network administrator to configure switches to add to the DCOD attached to the Research Network, we worked on bringing in more access to the Department Network by installing a spare L2/L3/L4 switch in the rack we had setup for the two groups whose systems all sat on the Department Network.

As we had anticipated, setting up a optical fiber link back to the Department Network did not work out – we only had 10 gigabit optical modules for our equipment and the Department network staff did not have spare 10 gigabit optical modules to support a fiber link to our switch, only 1 gigabit modules. Instead, we located a

free copper network jack in 420A, connected the switch to that, and proceeded to setup a pricey[6] L2/L3/L4 switch as the functional equivalent of a simple 48-port L2-only desktop switch. Though arguably a waste of an expensive piece of network hardware, this temporary loan allowed the department networking staff the flexibility to later provision a switch from their current vendor, get the proper optical modules, and configure it more fully to allow them to bring in multiple VLANs as needed.

At the end of the second week of May, we were ready to let the two groups move their systems into 420A. We wrote up instructions on how to get their systems up and running in 420A and distributed this the next Monday – they would be the first people to migrate out of 530. We would not be able to move our faculty's systems out of 530 until the third week of May due to delays in getting our network changes handled.

Network Lag

In the second week of May, we experienced slower progress with our network requests than we had expected. Though we estimated the actual work would only take an hour or two, we anticipated some delay due to forseeable factors on the part of the network administrator such as lack of familiarity with the Research Network, a strong desire to get everything setup exactly right for us, and an already busy work schedule. However, it became increasingly clear that the delay would be well beyond what we anticipated or could handle given the short timeframe.

Our first experienced a short delay when the network administrator pushed back a day on handling our network changes; though unexpected, we attributed it to a heavy workload due to taking on all day-to-day support for the Department Network after Network Guy's departure and felt that we could absorb this delay as we still had some facilities work left to finish.

Though the network administrator had worked with and trained on equipment from the vendor used by the Research Network, that experience had gone unused for a number of years due to the department's use of a different vendor. To help overcome this, one of our team wrote the switch configurations for the network administrator to load onto the switches, saving the network administrator time and transferring responsibility to us if something went wrong.

The final and unmistakable sign came near the end of the second week of May when we received notice that the reconfiguration of our switches, essentially loading the switch configurations we had written ourselves, had been reassigned to another member of staff who also had little to no recent experience with the network environment. It was at this point that we realized that something much more fundamental was going on here than just lack of time on the part of the network administrator. At this point, we only had a week left with all team members present to complete the work – any further delays would irreparably derail the migration.

A hour-long meeting with the IT Director revealed that he had been unaware of our previous access to manage the Research Network and of our relevant expertise and that he had also partly based his decisions on inaccurate information about relevancy and recency of other staff members' experience. We appreciated his desire to see both the Department and Research networks managed as a more unified single entity and understood his concerns that restoring our access to manage the Research Network could lead to further divergence, but we could no longer tolerate holding up the schedule to accomodate this nor wait for staff who weren't familiar with the systems involved. We reminded him of the lesson from from Brook's "The Mythical Man-Month"[1], namely that adding more staff to a project running behind schedule [or, as we added, running very close to it], particular staff unfamiliar with the project, will cause it to fall further behind schedule.

By the end of the meeting, we had, albeit with caveats about keeping the department network administrator informed, regained administrative access to the Research Network hardware and, with passwords literally in hand, we walked back up to 420A and started reconfiguring switches. That evening, we were up and running and ready to start moving our faculty's systems into 420A that night – on-schedule completion once again appeared within reach. In the weeks and months after the migration, we would return to the question of why it took so long to achieve our network goals, regardless of how we achieved them.

Back to Work

With our network problems resolved, we returned to the exodus from 530. With a few days of work time lost to dealing with the network management problems, we had to toss out a fair bit of our move schedule and start doing a lot more ad-hoc scheduling. Luckily, when we first started working out downtime schedules with users, instead of just setting specific dates, we also asked for "OK" windows, akin to space launch windows, during which it would be acceptable, though maybe not optimal, to shut a system down with short notice. This allowed us to aggressively pursue a more dynamic schedule where we check systems for user processes, quickly check-in with users about short downtimes that morning or afternoon, and, barring any objections, move systems in as large a batch as we could manage with downtimes hovering around an hour for each system. At times, users

would even proactively inform us that they were done with a system and could bear to be without use for some period of time, thus saving us the work of regular polling.

By the middle of the third week, we were back on track and even a little ahead of schedule as systems were being ferried down from 530 to 420A 3-4 times a day. By using a few spare and borrowed rack cabinets in 420A, we were able to more easily stage the migration of systems out of 530. The alternative of moving whole racks at a time (versus opportunistic migration of systems) would have been more challenging and would have allowed less opportunity for the reorganization of systems in 420A.

At the same time, we had managed to carve out time for one team member and the electrician to make a field trip down to Downtown Berkeley to confirm details of power and transport of the new faculty member's systems from IRB to EECS. By the end of the third week, we were down to just a few systems and lots of supplies and equipment left in 530 that were migrated to 420A or sent to storage. That weekend, one team member went out of town to handle site prep for an off-site research retreat followed by long-scheduled vacation and would not return to campus until two weeks later. While the IRB systems arrived on 25 May, we consider the day of completion for the migration to be 24 May which coincided with the first day of work for a newly hired computing Infrastructure Services Manager.

Lessons Learned

We learned a lot from this adventure. The one thing agreed upon by everyone involved as well as by several outside observers is a general sentiment of "Never. Again. Ever." This was probably near the limit of what could be accomplished with the resources we had available – and we came very close to failing. While we had options available for every problem we ran into, in some cases, some of those options had worse long-term consequences than failure. We got by on our own resourcefulness, persistence, and sheer luck to make up for a lack of room within which to fail. We hope that, by showing what happens when one runs a migration with the bare minimum time and staffing, others may take heed and step back from the proverbial cliff's edge that we danced upon for a period of a month.

A key overall lesson we kept coming back to was "Plans are just that – plans." The more time one has, the more one can try to bend reality to a plan – the less time one has, the more one must bend plans to fit reality. We lost track of the number of times some trivial problem arose that did not fit into our plan – like a supposedly 42RU high rack turning out to be 41.75RU high – instead of losing sleep over it, we found ways to adapt, like we did with the shorter-than-advertised rack by moving a server to another rack despite it not completely lining up with how we organized systems into racks. By looking at our plans more as initial guidelines or a starting point, we gave ourselves the freedom to deal with small unforseen problems without getting hung up about them and to look at larger problems not as problems but as course corrections.

The only truly disruptive problem was the delay in getting network changes made. While the problems we faced are getting a more formal review now that the new Infrastructure Services Manager is settled into his job, it is clear we had problems with communication, awareness of staff skillsets, and what could best be described as differences in culture. Others in the department have cited lack of project management, formal or otherwise, as a skill not explicitly present in our team – though our Facilities Director served as de facto project manager to keep track of progress and did communicate our problems to the IT Director, we found that the IT director did not understand the full situation until we had our own sit-down meeting with him. More explicit project management would likely have caught and handled these problems earlier. For our part, it is fair to say we could have been more direct and explicit about when we needed this work done and could have proactively listed restoration of our administrative access to the Research Network hardware as an alternative if a deadline were missed.

The question of "What would you have done if you had not received access?" has come up. We find it hard to believe that the IT Director would have said, "No." after being presented with the facts, but from a technical standpoint, it would have been easy – with physical access to the hardware, we could have the needed access via brute force methods.[3] This would have required disruptive power-cycling of each switch, but we could then do what was needed. From a management standpoint, it would have been contentious at best. At the very least, we would have informed our faculty and the Facilities Director of the problem and what we planned to do. It likely would also have perpetuated an image of us, whether rightly so or not, as "wild cannons" with little respect for authority and process who were difficult to work with and ultimately would have contributed to antagonistic relationships with our colleagues and our management. This was not something any of us would have considered casually.

Though not disruptive to the migration, still unfortunate were the moderate injuries suffered by staff. One team member, while working alone after hours with the raised floor, bashed his knee on two occasions, resulting in a bruised kneecap that took two months to heal completely. Had he suffered more serious injury, it was possible no one would have noticed until the next morning. His time working late would have been better spent

planning and preparing for the next day's work or even sleeping. Another staff member from Shipping and Receiving broke two fingers while moving into a elevator a cage rack that was still loaded onto a pallet. Though trained in use of pallet loaders, he was unaware that racks shipped on pallets often come with ramps to aid in unload the rack to avoid having to move the entire package like that – while most systems administrators in the department have had to deal with this and were aware of this feature, we wonder how widespread this knowledge is among facilities staff outside of those who work extensively in datacenters.

While attributable to the rapid pace and aggressive schedule of the migration, these injuries were avoidable indicating that at times some team members pushed themselves too hard or that they should have taken a more active role in physical tasks delegated to other staff to share our knowledge regarding better methods.

Our team took to heart the first lesson from "The Mythical Man-Month" about bringing on new team members to quick-paced or behind-schedule projects. All members had known each for at least a few years, had worked together on other projects, and were familiar with each other's habits as well as with the institutional knowledge of how things worked in the department. We did not bring our new systems administrator onto the migration project until after the majority of the facilities work had been finished and we could give him tasks that could be done jointly with someone else. We consider the case of assigning network configuration tasks to staff unfamiliar with the environment to be an example of the "Mythical Man-Month" fallacy that Brooks describes.

Similarly, we confirmed what might be considered an obvious notion – that fast-paced schedules are no place for in-depth staff training. When the schedule is fast, and the room for slack is tight, this is no place to be bringing in people new to the environment. The training overhead of new staff is unaffordably high in addition to the extra communication overhead of adding an extra person, experienced or not. As with our new guy, we instead followed another lesson from Brooks – give more routine tasks to new staff to allow more experienced staff to tackle the more difficult problems.

One thing that surprised us but really should have been obvious was the lack of convergence on any asychronous collabarative systems. Use of e-mail lists were sporadic at best, and only technical staff made any use of wikis or other web-based systems. As noted, for overall team synchronization, the only consistently successful method was a periodic standup meeting. Even among technical staff, paper notes left on racks often served better for passing short must-read notices about a rack or a system while things were in flux while the use of a wiki appeared more well suited to noting the steady state once a system or rack had been successful moved into place and deemed stable. This is extension of our previous lesson – once a project starts, training project members on new systems is inadvisable at best unless the benefit is so large compared to the training overhead that failure to adopt the new system could result in project failure.

The lessons we learned could best be summed up as "More time, better communication, and slow down." The ambitious plan to completely clean up the target room (420A) was cut short due to lack of time. While the result was a vast improvement, the time required to perform the cleanup was underestimated. In addition, it was unclear at the outset what parts (shelves, bolts, power strips, cables, etc.) might be needed for the final configuration. An earlier start on the clean-up would have helped; at least more of the parts could have been sorted in advance for subsequent disposition.

Future Work

While we were able to accomplish our basic goals of moving all systems from 530 to 420A in the time available, we still have a great deal of work left to bring 420A and other similar facilities in the department up to more contemporary standards. Some of this work is tractable in the near-term while other work will remain the focus of longer-term efforts involving questions of funding and staffing.

Our short-term work will focus on continual efforts at clean-up in all facilities. The one constant we have found about any datacenter facility, especially in an academic research environment, is the tendency for old equipment to accumulate and linger around so long that people forget what a piece of equipment is for and, as a result, become afraid to get rid of it. The one constant we have found about buildings with both raised floors and drop ceilings is the tendency for those areas to become absolutely filthy, especially raised floor plenums which cause systems to take in large amounts of dust and "biological debris". Both situations require both immediate and ongoing attention to mitigation. We are currently evaluating in the short-term a number of different options for front-of-rack filters.

Mid-term work focuses on networking, in particular the hack that is the 7-switch-long DCOD currently feeding almost all of the research systems in 420A. Current options include a 16-port 10GbE distribution/aggregation switch or setup of a double-ended string of switches. For now, the DCOD suffices, but the physical path of the fiber is tortuous at best due to the ad hoc manner in which the original stub network in 420A grew with fiber criss-crossing the room multiple times. Additional switches will only complicate this further and limit growth. Related to that is the replacement of the

switch currently deployed in a 'production' role on the Department Network. It should at the least be managed as a fully configured switch instead of setup as a $6,000 dumb L2 desktop switch, but ideally would be a model from the vendor currently used for the rest of the Department Network so that it can be more easily managed by department networking staff. Finally, we are very interested in finding ways to promote a more unified approach to network management that avoids the maintenance of two completely separate network domains.

Longer term work on the order of a year or more include several facilities upgrades. The first and foremost, but likely to take the longest is the replacement of the in-room CRACs which are nearly 20 years old and no longer supported by their manufacturer. Current measurements indicate that we are already using approximately 80% of our AC capacity while we still suffer a compressor failure about every 2-3 months in 420A. Replacement requres negotiation with our campus Physical Plant and could take a year if not more, but is neccessary to support future growth.

Other work we would like to pursue all relate to power distribution. The variety in types of circuits installed complicated work – standardizing on a single type of circuit, say 208VAC at 30A, would greatly ease management of the underfloor power distribution system and would make it easier to standardize on a single type or vendor of in-rack power distribution units (PDU). It is already hard enough to find PDUs with certain features – per outlet power monitoring, metering, and control, reasonably secure remote access, and a usable API for developing our own applications – that trying to account for even a handful of circuit types makes this impossible given current vendor offerings and impede efforts to gain better insight into power usage. We could pursue this work on a piecemeal basis, but we wonder if that would result in less standardization. Potentially the most extensive work we would like to pursue is transition to an overhead power distribution system. Though obviously costly, this would yield huge benefits in restoring orderly airflow to the underfloor air circulation space, easing of time costs of dealing with numerous circuits types and simplifying power usage surveys and audits.

On the non-technical side, we look forward to addressing clear deficits in key areas such as datacenter, network, and project management, communication and culture barriers between research and production operations, and management awareness of staff expertise along with staff awareness of management plans. We expect that this work will be ongoing for the rest of our professional careers – not because we think that we will always have these deficits but because the only way to avoid developing these sorts of deficits is by continually working to ensure that they do not develop.

Conclusions

We pulled it off, but just barely. We needed every single last day available in order to complete the work neccessary and could have used an extra day or two for breathing room. We got by on large measures of determination and dedication, resourcefulness, and sheer dumb luck. We look at this work as an accomplishment worth being proud of but also as an example illustrating all the things that one should have – many of which we did not – in order to embark on a similar datacenter migration adventure with a more reasonable chance of success and better options in case of something less than 100% success.

Acknowledgments

We would like to thank all the EECS staff who provided invaluable help without which this migration would not have happened, the faculty and researchers for their understanding of the pressures bearing down upon us and, of course, for all the caffeine that they made sure we had in ample supply.

References

[1] BROOKS, F. *The Mythical Man-Month.* Addison-Wesley, Reading, Massachusetts, 1975.

[2] CHA, L., MOTTA, C., , S., B., AND AGARWAL, M. What to do when the lease expires: A moving experience. In *Proceedings of LISA 1998* (1998).

[3] CISCO. Password recovery procedures. `http://www.cisco.com/en/US/products/sw/iosswrel/ps1831/products_tech_no%te09186a00801746e6.shtml`, Jan 2007.

[4] CUMBERLAND, D., HERBAN, R., IRVINE, R., SHUEY, M., AND LUISIER, M. Rapid parallel systems deployment: Techniques for overnight clustering. In *Proceedings of LISA 2008* (2008).

Notes

[1] Rough estimates for network closet relocation ran in $500K range.

[2] 1 ton of cooling is 12,000 BTU/hr or 3,517 W. The refrigeration and air conditioning fields use this unit to denote the heat required to melt 1 short ton, 2000 lbs, of ice at 0 °C in one day, representing the cooling provided by daily delivery of 1 ton of ice.

[3] 1 kVA is 1 kilovolt-ampere and is used to measure "Apparent Power", the product of root-mean-square voltage and current. "Real Power", measured in watts (W), refers to the power actually usable by devices.

[4] This "system" amounts to a grid of overhead Panduit-style fiber trays that meets up with an in-room fiber termination box fed by a nearby network closet.

[5] This trend would peak in the mid-2000s with the installation of a 128-node Itanium2 cluster.

[6] The switch was part of a large donation, so we do not know how much it cost the vendor to give it to us, but the current street price with optics is around $6000.

Bringing Up Cielo: Experiences with a Cray XE6 System

or, Getting Started with Your New 140k Processor System

Cory Lueninghoener cluening@lanl.gov
Daryl Grunau dwg@lanl.gov
Timothy Harrington toh@lanl.gov
Kathleen Kelly kak@lanl.gov
Quellyn Snead quellyn@lanl.gov

September 11, 2011

Abstract

High Performance Computing systems are complex to stand up and integrate into a wider environment, involving large amounts of hardware and software work to be completed in a fixed timeframe. It is easy for unforeseen challenges to arise during the process, especially with respect to the integration work: sites have dramatically different environments, making it impossible for a vendor to deliver a product that exactly fits everybody's needs. In this paper we will look at the standup of Cielo, a 96-rack Cray XE6 system located at Los Alamos National Laboratory. We will examine many of the challenges we experienced while installing and integrating the system, as well as the solutions we found and lessons we learned from the process.

Tags: HPC, configuration management, system integration

1 Introduction

High Performance Computing (HPC) systems are complex to stand up. They generally involve the delivery of a large amount of hardware at one time that must be installed, configured, and integrated in a fixed period of time in preparation to be run in relative steady state for five to ten years. While physical installation is usually a job for the hardware vendor, configuration and integration normally fall on the shoulders of a team of integrators and system administrators that will be charged with managing the system after it is put in production. Configuration and integration include such complex tasks as configuration management (CM) system design and implementation, parallel filesystem and network integration, and accounts system and user services integration. Standing up a new HPC resources can be similar to bringing up an entire new data center from scratch, with all of the same difficulties and pitfalls.

In this paper we will look at the installation and integration of Cielo, a 96-rack Cray XE6 compute resource sponsored by the Alliance for Computing at Extreme Scale (ACES) project and located at Los Alamos National Laboratory (LANL) and Sandia National Laboratory (Sandia). The complete Cielo family consists of four machines: Smog and Muzia, two 1/3-rack test systems; Cielito, a 1-rack development system; and Cielo, a 96-rack production system. Muzia is located at Sandia, while Smog, Cielito, and Cielo are located at LANL. In this paper we will focus on Cielo, but all four systems run the same software stack and have similar configurations, and our experiences with Cielo align with our experiences on the smaller systems.

2 System Overview

Cielo is a 96-rack, 142,304-core Cray XE6 system operated by the HPC Division at LANL. Delivery of the system began with the arrivals of Smog and Muzia at their respective laboratories in May of 2010, followed by delivery of Cielito in June of 2010 and 72 racks of Cielo in August of 2010. The final hardware delivery occurred in April of 2011, when Cielo was expanded to its full size of 96 racks. In this final configuration, Cielo consists of 8518 16-core compute nodes with 32GB of memory each; 376 16-core visualization compute nodes with 64GB of memory each; 286 IO nodes; 16 internal login nodes; and a handful of infrastructure nodes. Cielo debuted on the Top 500 List of Supercomputer Sites at position six in July of 2011.

The Cray XE6 is a massively parallel processing system that consists of compute and service blades connected by a high-speed torus network. The basic building block of the system is a *chassis* that holds eight compute or service blades. Each of these blades contains four diskless *nodes* that are designated as either *compute nodes*, where actual jobs run, or *service nodes*, which provide management, data virtualization service (DVS), or other services to the machine. Three chassis can be placed in a *rack*, and many racks can be combined in rows to create large systems. All of the nodes are connected by Cray's Gemini network, providing a three-dimensional torus high-speed network for user job communication, node management, and filesystem access.

All compute nodes are headless, diskless machines with no external network connections, but service nodes have one or two on-board PCI slots for expansion. These slots are most commonly used to provide outside network connections for login nodes or file server nodes. Two of the service nodes are designated as special: the *boot* node and the service database (*sdb*) node. These two nodes have external connections to a RAID device, the *bootraid*, and are used to provide persistent services to the rest of the nodes such as logging, root filesystems, and administrator management capabilities. One main task of the boot node is to serve the *sharedroot* from the bootraid to the other service nodes, who mount it as their root filesystem. In our case we also have a set of DVS nodes that serve this filesystem out to the compute nodes, giving them an optional complete Linux environment.

Outside of the main XE6 racks are two classes of standard rack-mounted machines: the system management workstation (SMW) and external login nodes. The SMW is the main administration server for the system, controlling low-level hardware access, system bootstrapping, and system ramdisks. The external login nodes are larger memory, diskfull analogs to the standard login nodes on the system blades that make user access more similar to standard clusters.

3 Challenges

Before pieces of Cielo even showed up at lab, the system administration team realized that we were going to have challenges integrating a Cray system into our environment. We currently run on the order of 20 HPC clusters, ranging from tiny (tens of nodes) up to huge (thousands of nodes), and we have been very careful to put configuration management at the forefront when installing new systems. All of our existing clusters are fully managed by Cfengine[1], with a collection of dedicated or multi-cluster CM servers covering every system in every cluster. This has tremendously helped our relatively small team keep all of the systems in sync and under control.

We quickly discovered that Cray has taken a "fully managed appliance" approach with the XE system. Part of this is due to the way the system itself is designed: the compute infrastructure

is tightly coupled by its torus network, with the compute and service node blade hardware fully managed by their control system. The compute nodes run a stripped down Linux-based operating system that is shipped as a Cray-provided image, while the service nodes run a Cray-branded release of SuSE's SLES 11 Linux distribution. Were we running Cielo as a stand-alone system, these features would all be fantastic: we could dedicate people to the system to keep it patched and running via Cray's methods, and that would be that. In fact, this appears to be a common way for sites to run large Cray systems: Cray will assign one or more software and hardware engineers to a site to handle much of the day-to-day management of the system, leaving the system administration team with more of a black box system to take care of.

Due to the existing usage models on our other systems, it was decided that we had a strong business case to not take the fully managed appliance route with Cielo. This way we could provide users with the most consistent environment across all of our systems while keeping our system administration team as flexible as possible. Instead, we needed to have our administrators integrate the system into our family of compute resources as tightly as we could. In our case, this was definitely the right decision: we've spent years keeping a wide variety of systems consistent to minimize the learning curve of existing users on new systems, and it has worked very well. However, it did present a series of challenges to the system integrators and administrators bringing up the systems. The challenges related to bringing up Cielo can be categorized into two broad areas: vendor relations and software.

3.1 Vendor Relations Challenges

The integration team that brought up Cielo was very fortunate to have a strong working relationship with Cray. The Cielo contract included provisions for Cray hardware, software, and application engineers to be located on-site at LANL, and through the bringup process we had access to many extra resources at Cray as we needed them. However, instead of just using the engineers as managers of the appliance, we worked with them to fully integrate the system with our already existing infrastructure. This being the first Cray system located at LANL in many years, there were some challenges on both sides as we worked out how we could work together most effectively.

Since we were looking to closely integrate Cielo into our environment, we ended up digging deeper into the inner workings of the machine than some of our Cray engineers were used to. In the process, we found that there was a lot of in-house knowledge at Cray that, although easy to get, was not always obvious to ask about. There were several times that we learned that a standard assumption about the management of the machine was incorrect, including such details as the best way to reboot nodes, the most effective way to correlate logs between the compute and service nodes, and how to update software. One of our running gags became the hypothetical response "Oh, you still do it that way?" to any question we asked of Cray.

On the bright side, we also had good challenges in this area: Cray provided written step-by-step procedures for almost everything we needed to do to the system. In many ways we weren't ready for this, especially when compared to the many commodity clusters we have in our environment. The biggest challenge here was figuring out how to make the best use of the documentation, whether it be importing the actions into Cfengine scripts, importing the documentation into our own library, or explicitly deciding to follow our own path. Cray was also very receptive to our suggestions for improvement. The challenges listed in the next few sections are often clearly related to our desire to do something our way instead of Cray's. However, they were ready to hear our suggestions and pass them on to their developers in most cases. There were several instance throughout Cielo's

bringup where specific nuances we found difficult were changed in later software updates.

3.2 Software Challenges

Over the course of the year that we have been working with Cielo, the majority of the challenges we have faced have come on the software front. As mentioned before, it is very common for Cray to assign a group of software engineers to a site with one of their large systems. We believe this practice has lead to many of our difficulties - Cray wasn't ready for us to be downloading and working with many of their software products, and we didn't have the experience needed to understand some of their distribution, packaging, and installation decisions.

3.2.1 Software Releases

There are three styles of software updates that we have worked with from Cray: cumulative service pack-style updates containing all previous updates for a particular product; individual patches and field notices that will eventually be rolled up in a cumulative update; and sliding window updates that contain older versions of the related software for compatibility as well as the latest update. Cumulative updates are generally released quarterly and contain new functionality, bug fixes, and other substantial updates. Individual patches and field notices are released as needed between the quarterly updates and generally fix bugs or security issues. Sliding window updates are used for the Cray programming environment and includes the both the latest and the last n releases of their supported compilers and libraries, where n is determined by Cray's release engineers based on provided functionality and customer usage. Each style of update is packaged differently, but each generally consist of a monolithic installer script, one or more directories full of RPMs, and fairly detailed instructions on how to install the update.

These individual release styles are further fragmented by individual update idiosyncrasies. Some updates are available publicly to all registered Cray users, while some are only available to Cray engineers. Most are applied by use of an included monolithic install script, but some are applied in a more manual process by following instructions in an install document. Some are applied in a way that will be preserved with future updates, while others are applied in a less stable way. Finally, versioning can be confusing: many packages include an SVN repository version number in their version string that refers to the repository revision from which it was generated. If branching happens in an unusual way, this can (and does) lead to newer software having a lower "version number" than already-existing software. All of these details are easily absorbed when the system is looked at as a standalone appliance, but in a more integrated environment that is used to a high level of update automation, they present a challenging hurdle.

3.2.2 Software Practices

Along with acclimating to Cray's software release methods described above, we ran into challenges with the software they contained. One of the biggest difficulties involved abuses of the RPM package management system. Most of the original software installs and subsequent updates provided by Cray comes in the form of RPM packages and monolithic install scripts that examine the hardware and software currently in use on the system and install required software as needed. This involves installing most packages with `--nodeps` and `--force` options that override RPM's built-in dependency and safety checks. Again, these options fit the appliance model very well: the supplied

software is vetted by Cray in a specific format, and they want to replicate that format closely in the field. However, they make verification and integration into an already-existing CM system difficult.

Similarly, Cray's distributed RPMs often make use of postinstall scripts to take care of large portions of the install. This use ranges from fairly benign (creating links if some other packages are already installed) to difficult to manage (RPM only contains a .tar.gz file that is unpacked by a postinstall script). These "write-only" RPMs also fit into the appliance model, but have many problems in a wider-ranging environment: they are difficult to verify, they can seem to behave non-deterministically depending on install order, and they tend to require more hand holding than more well-behaved packages. We found these packages especially difficult to place under CM control, as discussed in the next section.

Finally, we ran into several inconsistencies with respect to how software versions were managed. The `modules`[2] package and `/etc/alternatives` system[3] are two existing packages created to ease the selection of and switch between multiple versions of equivalent software installed on a system. The Cray software stack uses both of these packages to manage its software versions, even using both on the same package in some cases. In most cases they also use a third method involving "default links": symbolic links in each package's install path that point to the install root of the version that should be treated as default. These different methods are used to varying degrees by different pieces of the overall system - the modules are mostly used for normal users of the system to choose compilers, libraries, and related packages; while the alternatives and default links are mostly used by system-level processes. However, there is overlap between the usage of each.

3.2.3 Configuration Management

We discovered many of the challenges listed above while working to put Cielo under complete configuration management control. LANL's HPC division has traditionally used Cfengine to manage its systems, relying on it to create a uniform management environment across all of its clusters. Early on we recognized that this would be more difficult on Cielo than on traditional clusters, but we knew that it was the best way to integrate the new machine into our group's management rotation.

The software practices mentioned in the previous section all made configuration management challenging: creative uses of RPM, inconsistent versioning, and multiple management methods all add layers of complexity to the CM problem. However, the biggest challenge of all was Cray's use of monolithic install scripts. While very handy when installing software and updates interactively, these scripts made it difficult to automate configuration of the machine. Some of these scripts were easy to analyze and either import into Cfengine or call directly during the install process. Others were much more problematic: one explicitly checked to make sure it was connected to a TTY and exited with an error if it wasn't, while another stopped in the middle and presented a set of commands for the administrator to run in a second window before telling the script waiting in the first to continue. In another case, the script didn't even trust itself - after running, it instructs the administrator to check its work and confirm it had written out various files correctly. It turns out this was a needed step each time we ran it.

4 Responses

The above sections may sound like a large doom-and-gloom scenario, but in the end we were able to integrate Cielo into our environment in a way that is relatively easy for our administrators to pick up quickly. While we could have run the machine as a appliance-like system and not needed to deal with most of the challenges we discussed, we concluded that closer integration with our existing systems would help our small system administration team take on the system quickly after integration was complete. That made each of the challenges into actual problems and pushed us to find solutions for them.

4.1 Working Together

One of the most important things we did was keep a strong working relationship with Cray, our vendor. While it would be very easy for the clashes between our ideal system and their real-world products to result in deep fighting and animosity between the two groups, we were all able to keep a good relationship. Cray was eager to hear our concerns, fix problems, and submit idea cases when appropriate, and we were open to understanding their reasoning behind the design choices they made. We believe this is an important thing for both vendor and customer to keep in mind during a machine standup - both sides need each other, and keeping a good relationship is very beneficial in the long run.

Along with working closely with our vendor, we were careful to work closely across teams at LANL and Sandia. On the systems side, the Cielo bringup was a collaboration between HPC-5 (the system integrators) and HPC-3 (the system administrators) at LANL and the Cray support team at Sandia. Having these three teams work together gave us great power: we had the Cray experience from Sandia, the new system integration experience from HPC-5, and the long-term production system experience from HPC-3. While the nature of our environment made in-person collaboration the most useful form, we also made use of standard conference calls and email lists to keep each other in sync. The HPC-5/HPC-3 collaboration was especially important, as it made the transition from integration to production smooth and much less painless than a "throw it over the fence" model would have provided. Being able to work closely between all of the groups without chain of command overhead made it easy for us to make quick progress with the project.

4.2 Configuration Management

Another very important decision we made was to use a CM tool from the beginning. Although this could be seen as the source of several of our challenges, we would have had a much larger set of more difficult challenges without it. With Cielo, we ended up using a layered approach to managing the various parts of the system. Since the majority of the cluster is diskless, our final CM scheme had a small number of nodes that actually ran the Cfengine client: the SMW, the boot node, and the external login nodes. Everything else was managed by the sharedroot area (from the boot node) or the ramdisk images (from the SMW). With this design, we effectively had only a handful of Cfengine product areas to manage. This simplification made it easier to quickly grasp the design of the system and push out changes to the large number of nodes in the system.

Of course, after putting our CM system in place we still had a number of management tasks that required manual work. Most of these revolved around the monolithic install scripts mentioned previously - some of these were impossible to automate, while others just weren't worth the time.

For these we decided the best route was to document the exception and train new system administrators to recognize when they needed to do things by hand. In some cases, such as rebuilding the compute image ramdisk, we were able to have Cfengine print out a message after a successful run telling the administrator what more needed to be done by hand. In other cases we needed to rely on the carefully-maintained documentation wiki that the LANL administrators already use. By modeling the Cielo documentation off of existing documentation for other systems, we were able to fit these manual processes in to the mindset already known by the system administration team.

By implementing a complete configuration management scheme from the beginning, we were able to make several big changes to the system relatively quickly and painlessly. The first happened when we swung Cielo from our open network to our classified network: this required rebuilding the entire machine from scratch, which we were able to do in a matter of days using the configuration management system and documentation we had created. Later, we were able to quickly rebuild the system again when the upgrade from 72 to 96 racks required a large change in machine topology. Immediately after that, we made a quick upgrade between two Cray service packs that had caused problems at other sites with little trouble, mostly because we had a fully managed system and could recognize which system components had changed in incompatible ways. In short, our early effort has repaid itself several times over already.

4.3 Homegrown Tools

While bringing up Cielo, we found several system management deficiencies that weren't quite met by existing Cray management tools, but were too specialized to our environment to submit as a cases to Cray. Instead we wrote our own tools to fill in the gaps.

4.3.1 xtautorpm

Most of Cielo's compute and service nodes are diskless systems that use a ramdisk and an NFS-mounted read-only root filesystem to provide their operating system environment. The NFS filesystem provides system specialization of files through a layered approach, with the base filesystem being overlaid by *views* of node-specific files. These systems are managed by an interactive Cray tool named `xtopview` that handles package installation, file specialization, and other management tasks by presenting the administrator with a chrooted environment corresponding to the specialized view of each system or class of systems. This extra layer made our team's standard management methods difficult, as it is designed to be run interactively, only one person can run the `xtopview` utility at a time, and the utility has no provision for using tools such as `yum` to install packages.

To alleviate these restrictions, we expanded one of our already-existing tools under the name `xtautorpm`. This new tool automates acts as a layer between Cfengine and the `rpm` command, giving Cfengine the abstraction needed to use `xtopview` directly. With this extra layer of abstraction, we made the package installation procedure identical to that on our other clusters without losing the support of the vendor supplied tools.

4.3.2 xtfixdefault

As mentioned earlier, Cray uses several software version management schemes on their systems. We found it time consuming to manually manage both the `modules` environment (which we understood well) and the "default links" system that Cray introduced. To prevent version skew, we wrote a

tool named `xtfixdefault` to keep the two systems synchronized. Since we were already familiar with the `modules` system, we decided to use it as the base for our versioning. When Cfengine runs `xtfixdefault`, the utility checks all of Cray's default links and confirms that they are pointing to the same software versions as the `modules` environment's default version. When run interactively, the tool can also be used to update the modulefile from the default link and report on which modulefiles and default links are not the same. With this one utility we can both enforce our will over the software versions with Cfengine *and* report changes performed by Cray's monolithic software installers. This utility has made software updates much less time consuming.

4.3.3 ethcfg

The file specialization provided by the `xtopview` command is generally used at a class level to cover a large group of service nodes at once or at the individual node level to make one node stand out from the others. Both of these cases are simple and straightforward to manage. However, there is one specialization case that requires every node to have its own file: the static network management files. Standard configuration management systems avoid the need for hundreds of node-specific files by using templates, DHCP, or other similar solutions, but these did not fit the Cray model well. Instead we wrote a simple-but-powerful init script dubbed `ethtool` that configures the nodes' network interfaces by reading a flat configuration file at boot time. This file contains the network interface information for all of the service nodes in the system, meaning it can live in the default overlay view and requires no specialization for each node. The number of nodes included in the file is small enough that we found no performance problems with a flat file, giving us ease of maintainability over a more complex system using something like SQLite.

5 Lessons

After bringing up Cielo, we were able to put together a few lessons we learned along the way.

Keep good relations with your vendors : It is all too easy for vendor relations to break down when you don't see eye to eye with them. Keeping a good relationship makes it much easier to keep all sides progressing throughout the project.

Get test systems early : Although they were only mentioned briefly at the beginning of this report, our three smaller systems (Smog, Muzia, and Cielito) were instrumental in getting us experience with Cray's way of doing things early. When building a new system, getting access to representative hardware early in the process fills the knowledge pipeline much faster.

Use configuration management, even if it takes effort : The upfront cost of configuration management is easier to see than the long-term gains, but those gains are real. Whether you should work to fit a system into an existing CM scheme or not is a site-specific question, but using some tool is the best choice in any complex case.

When standing up a system of a new design, plan for "Murphy Time" : Murphy's Law will assert itself as often as it can, especially with new systems. Be ready for that. Finishing early is much more impressive than finishing late.

Work as a team : Today's systems are too complex for one person to fully understand. There are too may pieces: hardware, software, networking, filesystems, system management, and the list goes on. Working as a team is important for sharing responsibilities and areas of expertise with a new system and for keeping everybody interested in the project. Resist the urge to designate "the guy that knows it all", as he will inevitably win the lottery and leave the group.

6 Conclusions

As we stated in the beginning, standing up a new HPC resource is a complex task. While integrating Cielo, we ran into an expected breadth of challenges: managing the vendor/customer relationship, working with integrating an appliance-like system into an already-existing environment, designing a configuration management system around an imperfect software distribution design, and other more minor challenges. In our case these were all framed within the desire to make the system behave similarly to an already extensive set of HPC resources, a requirement from both the user and administrator points of view.

We were able to respond to these challenges with a combination of technical and social solutions involving, among more minor solutions, a close working relationship between our vendor and local teams, using strong configuration management and careful documentation when appropriate, and writing custom tools to fill in gaps as needed. The combination of solutions we found kept us flexible enough to make good decisions each time while getting the work done in the needed timeframe.

In the end, we were able to put together a short list of lessons that we thought were important from our experience. On the top of that list was the need to keep strong working relationships with all of the groups involved. Closely following this was the need for configuration management from the beginning. The list was rounded out with other lessons that are obvious in hindsight, but easy to lose track of in the heat of getting work done.

The final result of the work described in this report is a very manageable system. Like all systems of Cielo's complexity, there will always be work to be done, but we have a strong foundation on which to continue building and we are confident in the work we have done to integrate it into our environment.

References

[1] Cfengine. `http://www.cfengine.org/`.

[2] Modules – Software Environment Management. `http://modules.sourceforge.net/`.

[3] S. Kemp. Using the Debian alternatives system. `http://www.debian-administration.org/articles/91`.

Capacity Forecasting in a Backup Storage Environment

Mark Chamness
EMC
Mark.Chamness@emc.com

Abstract

Managing storage growth is painful [1]. When a system exhausts available storage, it is not only an operational inconvenience but also a budgeting nightmare. Many system administrators already have historical data for their systems and thus can predict full capacity events in advance.

EMC has developed a capacity forecasting tool for Data Domain systems which has been in production since January 2011. This tool analyses historical data from over 10,000 back-up systems daily, forecasts the future date for full capacity, and sends proactive notifications. This paper describes the architecture of the tool, the predictive model it employs, and the results of the implementation.

Tags: storage, predictive modeling, case study, capacity planning, forecasting, machine learning.

1 Introduction

Data storage utilization is continually increasing, causing the proliferation of storage systems in data centers. Monitoring and managing these systems requires increasing amounts of human resources and therefore automated tools have become a necessity.

IT organizations often operate reactively, taking action only when systems reach capacity, at which point performance degradation or failure has already occurred. Instead, what is needed is a proactive tool that predicts the date of full capacity and provides advance notification.

Predictive modeling has been applied to many fields: forecasting traffic jams [2, 3], anticipating electrical power consumption [4], and projecting the efficacy of pharmaceutical drugs [5]. Within the IT field, capacity management of server pools has been studied [6]. Ironically, there seems to be little previous work discussing applications of predictive modeling to data storage environments.

During the past year a predictive model has been employed internally at EMC to forecast system capacity and generate alert notifications months before systems reach full capacity. The ultimate purpose of this tool is to provide customers with both time and information to make better decisions managing their storage environment.

2 Data Collection

Data Domain systems are backup servers that employ inline deduplication technology on disk. All Data Domain back-up storage devices have a “phone-home” feature called Autosupport. Customers can configure their Data Domain systems to send an email every day with detailed diagnostic information. In addition, they can send email when specific events are encountered by the operating system. Once these emails are received at EMC, they are parsed and stored in a database.

Sending of diagnostic data via email to EMC is voluntary by the customer. Often, in secure environments, customers choose to disable the feature. In order to monitor their systems, customers have the ability to configure the autosupport emails to be delivered to internal recipients.

Most customers choose to send autosupports to EMC because the historical data enables more effective customer support. Given the more than 10,000 autosupports received daily, EMC has a statistically significant view across the Data Domain install base.

For the purpose of capacity forecasting, two variables are required at each point in time:

1. Total physical capacity of the system
2. Total physical space used by the system

For Data Domain systems, the total physical capacity changes over time because they generate an index which slightly decreases the amount of physical capacity available for data storage.

3 Data Cleaning

In order to ensure data integrity there are two issues to be addressed: data artifacts and elimination of non-production data.

Data Artifacts: In order to prevent bad data from entering the analysis, the tool assesses the quality of every autosupport and applies rules to guarantee consistency. These artifacts may arise due to an error in parsing the autosupport, data corruption during the transport of the autosupport, or both.

Non-Production Data: All internal Data Domain lab systems and QA systems send autosupports which are parsed and loaded into the database. These systems may be under development and therefore their performance characteristics may vary dramatically from production systems being used in the field. While this data is of value to internal teams, for the purposes of capacity forecasting, the data from these systems is excluded.

4 Predictive Model

One of the most common methods employed in predictive modeling is linear regression. Unfortunately, application of regression to storage capacity time series data is challenging because behavior changes. System administrators may add more shelves to increase capacity, change retention policies, or simply delete data. Therefore blind application of regression to the entire data set often leads to poor predictions.

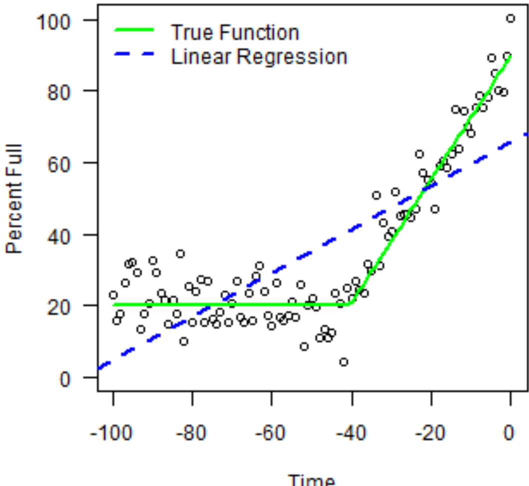


Figure 1: Example capacity data for the prior 100 days. (Time = 0 is the most recent data.) The standard deviation is 6 thoughout the data. The blue line shows the result of applying linear regression to the entire data set.

The predictions of the linear regression in Figure 1 are very poor. Intuitively, the data indicates the system is going to reach 100% capacity within a few days, but the regression line predicts far later (a false negative).

Select a Subset of Recent Data

The simplest method to mitigate the issue illustrated in Figure 1 would be to choose a subset of recent data such as the prior 30 days. This eliminates the influence of older data and improves the accuracy of the model's predictions. Unfortunately, using a fixed subset to model all systems results in poor linear models for many systems. Significantly more accurate models can be obtained by finding the optimal subset of data for each system and applying linear regression to only that subset of the data.

4.1 Piecewise Linear Regression

The error rate of the linear regression model can be significantly reduced by applying the regression to a data subset that best represents the most recent behavior. This requires implementing piecewise linear regression [7].

In order to find the best subset of data, the boundary must be determined where the recent behavior begins to deviate. The method described here analyses the quality of many linear regressions and then selects the one having the best fit.

The goodness-of-fit of a linear regression to experimental data can be measured by evaluating the coefficient of determination R^2. It is defined as the regression sum of squares ("SSM") divided by the total sum of squares ("SST") [8].

$$R^2 = \frac{SSM}{SST} = \frac{\sum_i [f(x_i) - \bar{y}]^2}{\sum_i [y_i - \bar{y}]^2}$$

Properties of R^2

- $0 \leq R^2 \leq 1$
- $R^2 = 1$ indicates perfectly linear data.

Calculating the Boundary: Start with a small subset of the data, such as the prior 10 days, and then apply regression to incrementally larger subsets to find the regression having the maximum value of R^2.

1. Regress $\{(x_{-10}, y_{-10}), (x_{-9}, y_{-9}), \ldots, (x_0, y_0)\}$
2. Calculate R^2 for regression
3. Regress $\{(x_{-11}, y_{-11}), (x_{-10}, y_{-10}), \ldots, (x_0, y_0)\}$
4. Calculate R^2 for regression
5. …
6. Regress $\{(x_{-n}, y_{-n}), (x_{-n+1}, y_{-n+1}), \ldots, (x_0, y_0)\}$
7. Calculate R^2 for regression
8. Select the subset with maximum R^2

The boundary is the oldest data point within the subset of data determined in step 8. The predictive model is generated by applying linear regression to that subset.

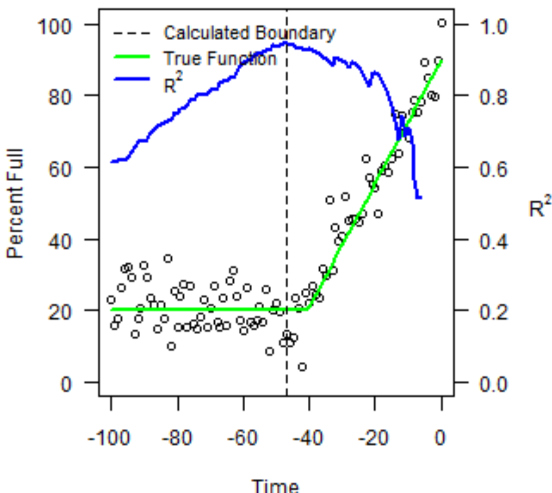


Figure 2: The same data used in Figure 1 with R^2 plotted for each subset of data. The date when R^2 reaches its maximum value is the "calculated boundary" and occurs near the discontinuity of the true function. Maximum $R^2 = 0.95$ at -48 days and the true boundary is -40 days.

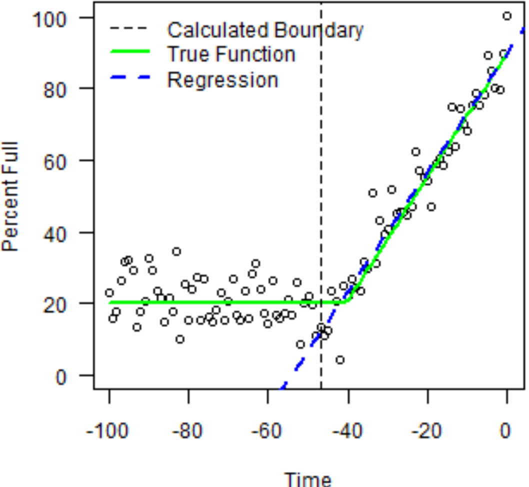


Figure 3: The same data from Figures 1 & 2. Piecewise linear regression results in a better fit to the data. This model was generated using the subset $\{(x_{-48}, y_{-48}), \ldots, (x_0, y_0)\}$ determined by the boundary.

Preprocessing data by applying a smoothing function can increase R^2, but has limitations. Filtering out noise while maintaining the signal is easier said than done. Too much smoothing and it becomes too difficult to determine the boundary point.

4.2 Other Models

Many other models can be applied to time series data, such as weighted linear regression, logarithmic regression, and auto-regressive (AR) models. In the current implementation, a simple linear model has shown to effectively model many systems (see Section 5: "Results of Predictive Modeling"). It is an open question whether the remaining systems can be modeled by other methods.

4.3 Model Validation

The model needs to be able to say, "I don't know." Sometimes there is no pattern in the data. Before employing a model to predict future behavior, it should be evaluated to determine if it is reasonable model for the data set. In the current implementation, validation rules are applied to the results of the linear model to determine if capacity forecasts should be published.

Goodness-of-fit: When the R^2 value from piecewise linear regression is too small, it indicates the model is a poor fit to the data. In the current tool, linear regression models with $R^2 < 0.90$ are not used.

Positive Slope: Linear models having a zero or negative slope cannot be used to predict the date of 100% full.

Timeframe: Forecasts for systems to reach full capacity far into the future are extrapolating the current behavior too much to be practical. The current model limits forecasts to less than 10 years. The expectation is that within 10 years the storage technology will be significantly different than it is today.

Sufficient Statistics: Storage systems that have been recently deployed lack enough historical data to produce statistically sufficient regression models. A minimum of 15 days of data is a reasonable threshold for the size of the data set.

Choosing a smaller minimum value may result in fitting the model to noise. Linear regression can achieve a very good fit to a handful of data points, but the results are not statistically significant.

Space Utilization: Experience has shown that systems which are less than ~10% full tend not to produce reliable predictions. For this situation, the current tool does not generate capacity forecasts.

Last Data Point Trumps All: Recent changes in system capacity must be taken into account to evaluate the linear fit. When systems are nearing maximum storage capacity, the administrator often takes action which results in drastic changes in the amount of available capacity. If the administrator reduces the amount of data stored on a device, the capacity prediction of the model is no longer valid. Assessing this error is a simple form of cross-validation.

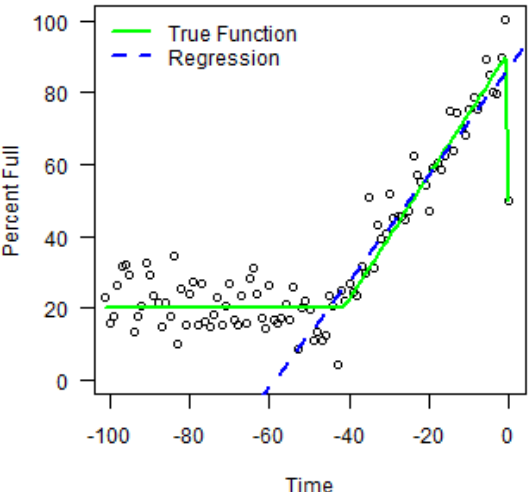


Figure 4: System capacity dropped from 100% to 50%. The generated linear model has a high goodness-of-fit ($R^2 = 0.89$) but the prediction for the most recent data point has 35% percent error. The model predicts the system is 85% full at Time=0, but it is only 50% full.

If the error between the predicted value and the actual value of the most recent data point exceeds 5%, it is a good indication that the recent data diverges significantly from the model and therefore the model is no longer valid.

5 Results of Predictive Modeling

5.1 Analysis of Linear Regression Fit to Past Data

If historical data does not demonstrate linear growth, then obviously linear regression would be a poor model to employ. To investigate this issue, the piecewise linear regression algorithm described in section 4.1 was applied to the historical dataset from Data Domain storage appliances and the maximum R^2 was calculated for each system.

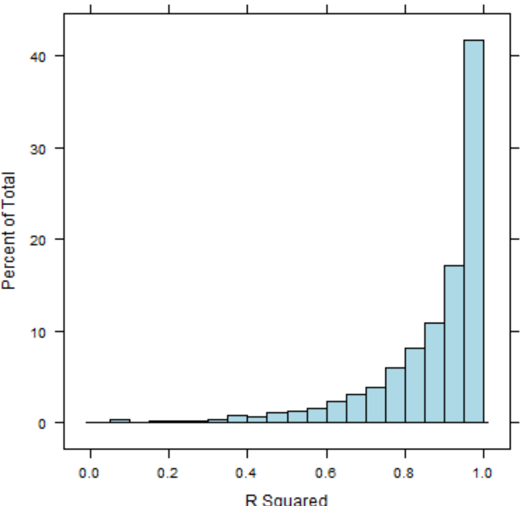


Figure 5: Histogram of R^2 across all systems using a minimum 15 days of data. This illustrates that most of the regression models generated for storage systems have R^2 close to 1.0.

Summary of results:

1. The median R^2 for all systems was 0.93
2. Models for 60% of systems had $R^2 \geq 0.90$
3. Models for 78% of systems had $R^2 \geq 0.80$

These results indicate that the majority of systems exhibit very linear behavior since the linear model had a very good fit to the datasets.

5.2 Forecasting Full Capacity

After the model is generated from historical data, the next step is to apply the validation rules described in section 4.3. For models that pass validation, the final step is to solve for the future date the system will become 100% full. The linear model:

$$y = \alpha + \beta x$$

Definitions:

- y is capacity
- α is the intercept term
- β is the slope
- x is the date

Assuming the slope is positive (β >0), the future date for the system reaching full capacity can be calculated by setting the capacity y = 1 (100 %) and solving for x:

$$\text{Forecast Full Date: } x = \frac{1-\alpha}{\beta}$$

5.3 Analysis of the Quality of Forecasts

False Positives

False positives frequently originate from unforeseen future human activities which cannot be predicted by the model. It is difficult to construe such false positives as flaws in the model per se given that the only input provided to the model is historic behavior of the system.

When a system is on a linear trajectory to full capacity but never reaches 100% full, it is may be due to external or internal events. An external event may originate from a significant change in the amount or rate of data placed into primary storage. An event internal to the system may be caused by the system administrator taking action to implement configuration changes. These can include:

1. Hardware changes
   a. The system was entirely replaced
   b. A shelf was added , increasing capacity
   c. Internal disk drives were replaced
2. Software changes
   a. Retention policy was changed
   b. Data was deleted and/or moved

A specific example may help elucidate the issues concerning false positive capacity forecasts. Even with visual inspection of the data by a human, it is extremely difficult to assess a false positive a priori.

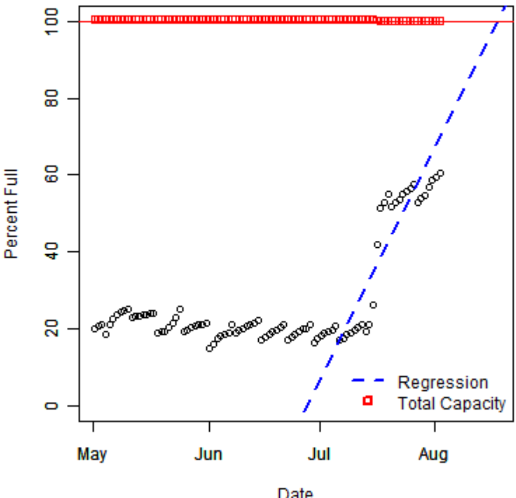


Figure 6: System exhibiting several changes in the rate of storage utilization. At this point in time the regression may be a *false positive*.

Visual inspection of the storage capacity of the system in Figure 6 indicates that the rate of storage is on a trajectory to reach full capacity in September. However, the most recent data in August might be an early indication that the trajectory is changing. This recent data may imply the system is stabilizing near 60% of capacity, but at this point in time there is insufficient data to establish a new trajectory.

From a statistical perspective, it is unknown whether the recent data points are signal or noise. This illustrates how allowing the use of small data sets has the risk of fitting the model to noise.

Ironically, in spite of the intuitive uncertainty, the fit to data is very good: R^2 = 0.90 and the prediction error is only 4.5% on the most recent data point. This example is potentially a good candidate for the model to fail validation and report, "I don't know." There is a trade-off between eliminating reasonable models versus generating false positives. By requiring more data for models, we gain higher confidence in their predictions, but reduce the advanced notification for true positives.

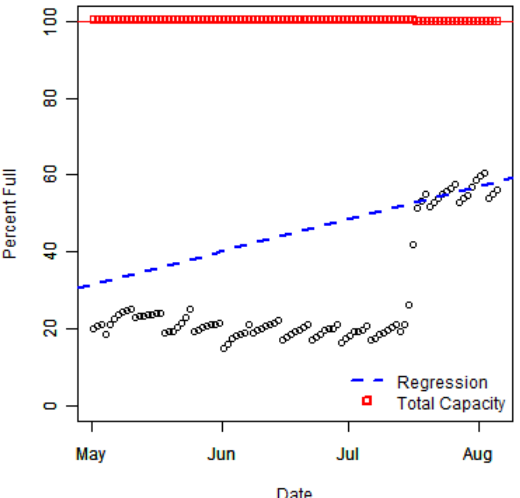


Figure 7: Same system shown in Figure 6 with additional data points.

After a few more days, the piecewise regression model fits the recent behavior of the system in Figure 7. Only after obtaining more data can we determine that the model in Figure 6 was a false positive. It is often the case that false positives can only be observed with the benefit of hindsight (addition data).

No Forecast for Full Capacity

When a model fails validation (described in section 4.3) no forecast should be made. On a typical day the current model does not publish forecasts for approximately 40% to 50% of all systems. This is not a surprising result. Most systems are expected to be efficiently managed by their administrators. The model is only considered valid for systems which are on a trajectory to full capacity in the future.

It is an operational decision to determine the quantity of forecasts to be published. The percent of systems for which forecasts are published can be easily adjusted by tailoring the validation rules for each environment.

5.4 Analysis of Forecasts across Install Base

Application of the model described to the entire install base results in a number of observations.

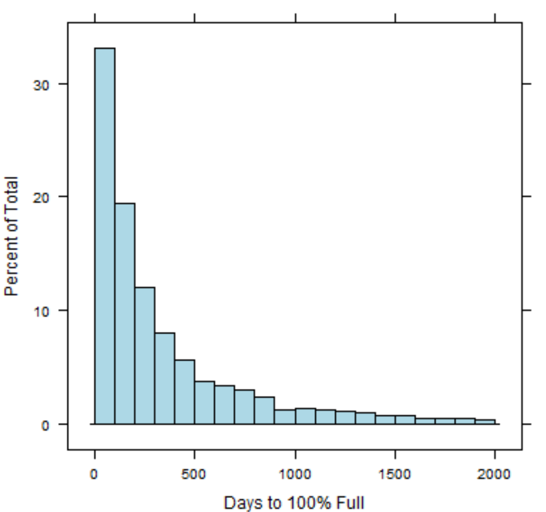


Figure 8: Histogram of forecasts for systems to reach full capacity. The median time to 100% full is 197 days. Therefore, for systems with valid models, the forecast is half of them will reach full capacity within approximately six months.

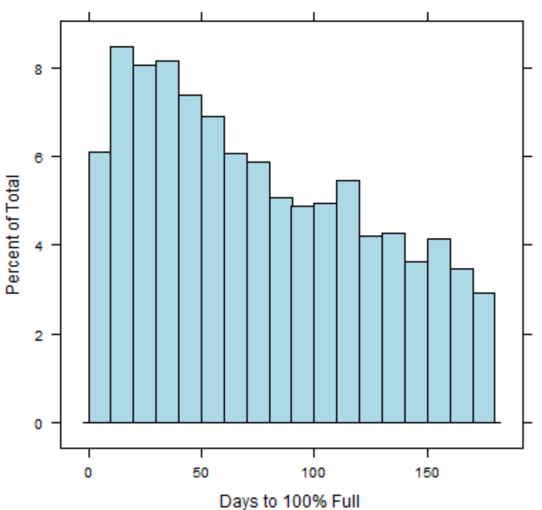


Figure 9: Greater detail (6 months) of the data used in Figure 8.

Given the peak values of these histograms, a majority of the systems are predicated by the model to reach full capacity in the near future. There are at least two conjectures that may explain the patterns of Figures 8 & 9:

Hypothesis 1: Efficient use of capital: Since the cost of storage (dollars per GB) drops quickly over time, the majority of storage devices are intended to only have enough space for the near future. It's cheaper to delay the purchase of additional storage until it's absolutely needed.

Hypothesis 2: Capacity Exceeded Expectations: System administrators forecasted their capacity needs for the long-term, but they underestimated the rate of growth.

6 Capacity Forecasting Examples

The application of capacity forecasting may be illustrated by examining a few examples of production Data Domain storage systems.

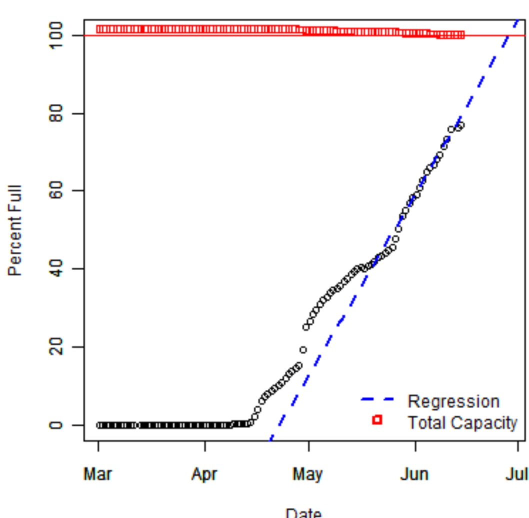


Figure 10: System exhibiting linear segments.

This type of behavior was the motivation for developing the piecewise linear regression algorithm. The data prior to May is useless for prediction since it significantly different from the current behavior of the system. Application of piecewise linear regression correctly found a model that fits the data from the beginning of June to the last data point.

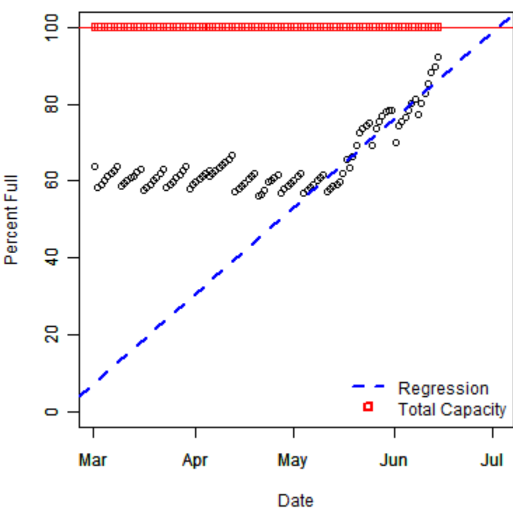


Figure 11: A behavioral change in the rate of storage utilization occurred at the end of May, but the piecewise linear regression model correctly fit the most recent behavior despite noisy data.

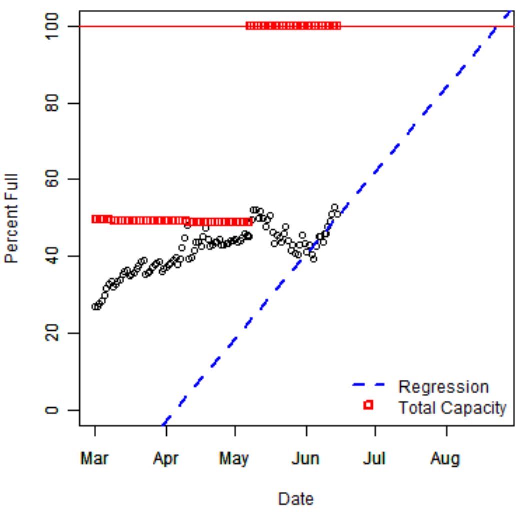


Figure 12: Shelf was added to an existing Data Domain system.

The total capacity exhibits a discontinuity in May. In this figure, the system reached full capacity and then a shelf was added. The model fits the recent data and predicts the system will reach 100% capacity in approximately three months.

7 Conclusions and Future Work

The role of automated predictive modeling for managing IT systems will become more pervasive as the complexity and size of data centers continue to grow. [9]

This paper describes a model that uses historical data to predict when Data Domain systems will reach full capacity. Advance notice of storage systems reaching full capacity allows system administrators to take necessary measures to avoid performance degradation and/or failure. It was demonstrated that many storage systems can be modeled using a piecewise linear regression model. Furthermore it was shown that for the systems that could be modeled, they were able to generate a forecast of the date of full capacity in advance.

Many questions still remain for future analyses which are natural extensions of the material discussed in this paper:

1. Are there other applications of predictive modeling within the existing data set? Could compression ratio, bandwidth throughput, load-balancing [10] or IO capacity also be predicted?
2. Why was the piecewise linear regression model not able to model some systems? Could the model be improved or could they be modeled by some other method?
3. Using the statistically significant view across the install base, could there be correlations between system variables or time series correlations for a single variable?

Capacity forecasting is a fundamental utility for system management, but it is only a starting point of the data analysis that can be explored for storage management.

Acknowledgments

I thank Stephen Manley, Fred Douglis, and Philip Shilane for both guidance and meticulous comments on early drafts. I especially appreciate the feedback and direction provided by my shepherd, Andrew Hume, as well as the anonymous referees.

8 References

[1] TheInfoPro. "Deduplication: A paradigm Shift in Backup", *TheInfoPro (TIP) Research Paper*, January 2011. Available at: https://community.emc.com/docs/DOC-9720

[2] Andras Hegyi. "Model Predictive Control for Integrating Traffic Control Measures", February 2004. Available at: http://www.dcsc.tudelft.nl/~deschutt/research/phd_theses/phd_hegyi_2004.pdf

[3] Eric Horvitz, Johnson Apacible, Raman Sarin, and Lin Liao. "Prediction, Expectation, and Surprise: Methods, Designs, and Study of a Deployed Traffic Forecasting Service", *Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI-2005, Edinburgh, Scotland, July 2005. Available at: http://research.microsoft.com/~horvitz/horvitz_traffic_uai2005.pdf

[4] Eduardo Camponogara, Dong Jia, Bruce H. Krogh, and Sarosh Talukda. "Distributed Model Predictive Control". *Control Systems, IEEE*, February 2002. Available at: http://www.ece.cmu.edu/~krogh/papers/CJKT02.pdf

[5] WM Watkins, EK Mberu, PA Winstanley. "The efficacy of antifolate antimalarial combinations in Africa: a predictive model based on pharmacodynamic and pharmacokinetic analyses", *Parasitology Today, Volume 13, Issue 12*, December 1997, Pages 459-464, 1997

[6] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, Alfons Kemper, "Capacity Management and Demand Prediction for Next Generation Data Centers," icws, pp.43-50, IEEE International Conference on Web Services (ICWS 2007), 2007

[7] Robert Nisbet, John Elder, and Gary Miner. *Statistical Analysis and Data Mining Applications*. Academic Press, 2009

[8] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Pearson Education, 2006

[9] IBM. "The growing role of predictive analytics in data center management", *IBM Developer Works*, December 2010. Available at: https://www.ibm.com/developerworks/mydeveloperworks/blogs/business-analytics/entry/the_growing_role_of_predictive_analytics_in_data_center_management

[10] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, Philip Shilane. "Content-aware Load Balancing for Distributed Backup", LISA 2011: Proceedings of the 25th Large Installation System Administration Conference (Dec 2011)

Content-aware Load Balancing for Distributed Backup

Fred Douglis
EMC
Fred.Douglis@emc.com

Deepti Bhardwaj
EMC
Deepti.Bhardwaj@emc.com

Hangwei Qian*
Case Western Reserve Univ.
Hangwei.Qian@case.edu

Philip Shilane
EMC
Philip.Shilane@emc.com

Abstract

When backing up a large number of computer systems to many different storage devices, an administrator has to balance the workload to ensure the successful completion of all backups within a particular period of time. When these devices were magnetic tapes, this assignment was trivial: find an idle tape drive, write what fits on a tape, and replace tapes as needed. Backing up data onto deduplicating disk storage adds both complexity and opportunity. Since one cannot swap out a filled disk-based file system the way one switches tapes, each separate backup appliance needs an appropriate workload that fits into both the available storage capacity and the throughput available during the backup window. Repeating a given client's backups on the same appliance not only reduces capacity requirements but it can improve performance by eliminating duplicates from network traffic. Conversely, any reconfiguration of the mappings of backup clients to appliances suffers the overhead of repopulating the new appliance with a full copy of a client's data. Reassigning clients to new servers should only be done when the need for load balancing exceeds the overhead of the move.

In addition, deduplication offers the opportunity for *content-aware* load balancing that groups clients together for improved deduplication that can further improve both capacity and performance; we have seen a system with as much as 75% of its data overlapping other systems, though overlap around 10% is more common. We describe an approach for clustering backup clients based on content, assigning them to backup appliances, and adapting future configurations based on changing requirements while minimizing client migration. We define a cost function and compare several algorithms for minimizing this cost. This assignment tool resides in a tier between backup software such as EMC NetWorker and deduplicating storage systems such as EMC Data Domain.

*Work done during an internship.

Tags: backups, configuration management, infrastructure, deduplication

1 Introduction

Deduplication has become a standard component of many disk-based backup storage environments: to keep down capacity requirements, repeated backups of the same pieces of data are replaced by references to a single instance. Deduplication can be applied at the granularity of whole files, fixed-sized blocks, or variable-sized "chunks" that are formed by examining content [12].

When a backup environment consists of a handful of systems (or "clients") being backed up onto a single backup appliance (or "server"), provisioning and configuring the backup server is straightforward. An organization buys a backup appliance that is large enough to support the capacity requirements of the clients for the foreseeable future, as well as capable of supporting the I/O demands of the clients. That is, the backup appliance needs to have adequate *capacity* and *performance* for the systems being backed up.

As the number of clients increases, however, optimizing the backup configuration is less straightforward. A single backup administration domain might manage thousands of systems, backing them up onto numerous appliances. An initial deployment of these backup appliances would require a determination of which clients to back up on which servers. Similar to the single-server environment, this assignment needs to ensure that no server is overloaded in either capacity or performance requirements. But the existence of many available servers adds a new dimension of complexity in a deduplicating environment, because some clients may have more content in common than others. Assigning similar clients to the same server can gain significant benefits in capacity requirements due to the improved deduplication; in a constrained environment, assigning clients in a content-aware fashion can make the difference between meeting

one's capacity constraints and overflowing the system.

The same considerations apply in other environments. For example, the "clients" being backed up might actually be virtual machine images. VMs that have been cloned from the same "golden master" are likely to have large pieces in common, while VMs with different histories will overlap less. As another example, the systems being copied to the backup appliance might be backup appliances themselves: some enterprises have small backup systems in field offices, which replicate onto larger, more centralized, backup systems for disaster recovery.

Sending duplicate content to a single location can not only decrease capacity requirements but also improve performance, since content that already exists on the server need not be transferred again. Eliminating duplicates from being transmitted is useful in LAN environments [5] and is even more useful in WAN environments.

Thus, in a deduplicating storage system, *content-aware* load balancing is desirable to maximize the benefits of deduplication. There are several considerations relating to how best to achieve such balance:

Balancing capacity and throughput Above all, the system needs to assign the clients in a fashion that minimizes hot spots for storage utilization or throughput. Improvements due to overlap can further reduce capacity requirements.

Identifying overlap How does the system identify how much different clients have in common?

Efficiency of assignment What are the overheads associated with assignment?

Coping with overload If a server becomes overloaded, what is the best way to adapt, and what are the costs of moving a client from that server?

Our paper has three main contributions:

1. We define a *cost function* for evaluating potential assignments of clients to backup servers. This function permits different configurations to be compared via a single metric.

2. We present several techniques for performing these assignments, including an iterative refinement heuristic for optimizing the cost function in a content-aware fashion.

3. We compare multiple methods for assessing content overlap, both for collecting content and for clustering that content to determine the extent of any overlap.

Our assignment algorithm serves as a middleware layer that sits between the backup software and the underlying backup storage appliances. Our ultimate goal is a fully automated system that will dynamically reconfigure the backup software as needed. As an initial prototype, we have developed a suite of tools that assess overlap, perform initial assignments by issuing recommendations for client-server assignments, and compute updated assignments when requirements change. Client assignments can be converted into a sequence of commands to direct the backup software to (re)map clients to specific storage appliances.

The rest of this paper is as follows. The next section provides more information about deduplication for backups and other related work. §3 provides use cases for the tool. §4 describes load balancing in more detail, including the "cost" function used to compare configurations and various algorithms for assignment of clients to servers. §5 discusses the question of content overlap and approaches to computing it. §6 presents results of simulations on several workloads. §7 examines some alternative metrics and approaches. Finally, §8 discusses our conclusions and open issues.

2 Background and Related Work

2.1 Evolving Media

In the past decade, many backup environments have evolved from tape-centric to disk-centric. Backup software systems, such as EMC NetWorker [6], IBM Tivoli Storage Manager [9], or Symantec NetBackup [19], date to the tape-based era. With tapes, a backup server could identify a pool of completely equivalent tape drives on which to write a given backup. When data were ready to be written, the next available tape drive would be used. Capacity for backup was not a critical issue, since it would usually be simple to buy more magnetic tape. The main constraint in sizing the backup environment would be to ensure enough throughput across the backup devices to meet the "backup window," i.e., the time in which all backups must complete. Some early work in this area includes the Amanda Network Backup Manager [16, 17], which parallelized workstation backups and created schedules based on anticipated backup sizes. Interleaving backup streams is necessary to keep the tapes busy and avoid "shoe-shining" from underfull buffers, but this affects restore performance [20]. The equivalence of the various tape drives, however, made parallelization and interleaving relatively straightforward.

Disk-based backup grew out of the desire to have backup data online and immediately accessible, rather than spread across numerous tapes that had to be located, mounted, and sequentially accessed in case of data loss. Deduplication was used to reduce the capacity requirements of the backup system, in order to permit disk-

based backup to compete financially with tape. The most common type of deduplication breaks a data stream into "chunks," using features of the data to ensure that most small changes to the data do not affect the chunk boundaries. This way, inserting a few bytes early in a file might change the chunk where the insertion occurs, but the rest of the file will deduplicate. Deduplicating systems use a strong hash (known as a "fingerprint") of the content to identify when a chunk already exists in the system [15, 21].

2.2 Deduplication: Challenges and Opportunities

With deduplicated disk backups replacing tape, the equivalence of appliances is partly lost. Writing to the same storage system gains efficiencies by suppressing duplicate data; these efficiencies can be further reflected back to the backup server or even the client being backed up, if the duplicates are identified before data cross the network [5]. The effort of dividing the content into chunks and computing fingerprints over the chunks can be distributed across the backup infrastructure, allowing the storage appliance to scale to more clients and reducing network traffic when the deduplication rate is high.

Thus, the "stickiness" of the assignment of a client to a storage appliance changes the role of the backup administrator. Instead of simply pooling many clients across many tape drives, the mapping of clients to storage appliances needs to be done *a priori*. Once a client has been paired with a particular storage appliance, it gets great benefits from returning to that appliance and omitting duplicates. Should it move to a different appliance, it must start over, writing all of its data anew. But if its target appliance is overloaded, the client queues up and waits longer than desired, possibly causing the backup not to complete within its "backup window." Capacity is similarly problematic, since a client that is being backed up onto a full storage appliance either is not protected or must move to another less loaded system and pay a cost for copying data that would otherwise have been suppressed through deduplication.

In summary, once a client is backed up onto a particular storage appliance, there is a tension between the benefits of continuing to use it and the disadvantages that may ensue from overload; at some tipping point, the client may move elsewhere. It then pays a short-term overhead (lack of deduplication) but gets long-term benefits.

Another interesting challenge relating to deduplicating storage is anticipating when it will fill up. One needs to consider not only how much is written but also how well that data will deduplicate. Predictions of future capacity requirements on a device-by-device basis, based on mining past load patterns [2], would feed into our load balancing framework.

2.3 Load Balancing

Finally, the idea of mapping a set of objects to a set of appropriate containers is well-known in the systems community. Load balancing of processor-intensive applications has been around for decades [3, 8], including the possibility of dynamically reassigning tasks when circumstances change or estimates prove to be inaccurate [13]. More recently, allocating resources within grid or cloud environments is the challenge. Allocating virtual resources within the constraints of physical datacenters is particularly problematic, as one must deal with all types of resources: processor, memory, storage, and network [18]. There are many examples of provisioning systems that perform admission control, load balancing, and reconfiguration as requirements change (e.g., [7]), but we are unaware of any work that does this in the context of deduplication.

3 Use Cases

In this section we describe the motivation behind this system in greater detail. Figure 1 demonstrates the basic problem of assigning backups from clients to deduplicated storage systems, and there are a number of ways in which automated content-aware assignment can be useful.

Sizing and deployment Starting with a "clean slate," an administrator may have a large number of client machines to be backed up on a number of deduplicating storage appliances. The assignment tool can use information about the size of each client's backups, the throughput required to perform the backups, the rate of deduplication within each client's backups, the rate at which the backup size is expected to change over time, and other information. With this data it can estimate which storage appliances will be sufficient for this set of clients. Such "sizing tools" are commonplace in the backup industry, used by vendors to aid their customers in determining requirements. Using information about overlapping content across clients allows the tool to refine its recommendations, potentially lowering the total required storage due to improved deduplication.

First assignment Whether the set of storage appliances is determined via this tool or in another fashion, once the capacity and performance characteristics of the storage appliances are known, the tool can recommend which clients should be assigned to which storage system. For the first assignment, we assume that no clients are already backed up on any storage appliance, so there is no benefit (with respect to deduplication) to preferring one appliance over another for

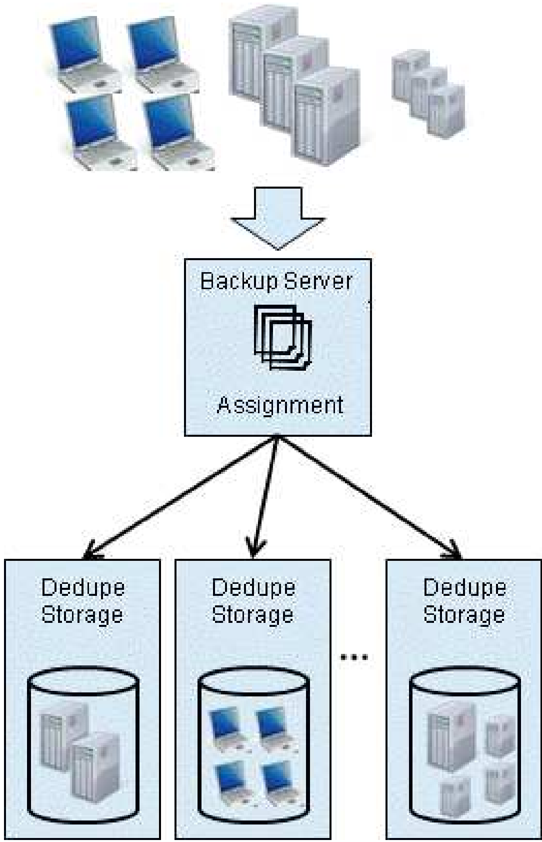


Figure 1: Backups from numerous clients are handled by the backup manager, which assigns clients to deduplicated storage while accounting for content overlap between similar clients. In this case, there was insufficient room on a single storage node for all three large servers, so one was placed elsewhere.

individual clients, though overlapping content is still a potential factor.

Reconfigurations Once a system is in steady state, there are a number of possible changes that could result in reconfiguration of the mappings. Clients may be added or removed, and backup storage appliances may be added. Storage may even be removed, especially if backups are being consolidated onto a smaller number of larger-capacity servers. Temporary failures may also require reconfiguration. Adding new clients and backup storage simultaneously may be the simplest case, in which the new clients are backed up to the new server(s). More commonly, extra backup capacity will be required to support the growth over time of the existing client population, so existing clients will be spread over a larger number of servers.

Disaster recovery As mentioned in the introduction, the "clients" might be backup storage appliances themselves, which are being replicated to provide disaster recovery (DR). In terms of load balancing, there is little distinction between backing up generic computers (file servers, databases, etc.) and replicating deduplicating backup servers. However, identifying content overlap is easier in the latter case because the content is already distilled to a set of fingerprints. Also, DR replication may be performed over relatively low-bandwidth networks, increasing the impact of any reconfiguration that results in a full replication to a new server.

4 Load Balancing and Cost Metrics

In order to assign clients to storage appliances, we need a method for assessing the relative desirability of different configurations, which is done with a *cost* function described in §4.1. Given this metric, there are different ways to perform the assignment and evaluate the results. In §4.2, we describe and compare different approaches, both simple single-pass techniques that do not explicitly optimize the cost function and a heuristic for iteratively minimizing the cost.

4.1 Cost Function

The primary goal of our system is to assign clients to backup servers without overloading any individual server, either with too much data being stored or too much data being written during a backup window. We define a *cost metric* to provide a single utility value for a given configuration. The cost has several components, representing skew, overload, and movement, as shown in Table 1.

The basic cost represents the skew across storage and throughput utilizations of the various servers, and when the system is not heavily loaded it is the dominant component of the total cost metric. Under load, the cost goes up dramatically. Exceeding capacity is considered fatal, in that it is not a transient condition and cannot be recovered from without allocating new hardware or deleting data. Exceeding throughput is not as bad as exceeding capacity, as long as there is no "hard deadline" by which the backups must complete — in that event, data will not be backed up. Even if not exceeded, the closer capacity or throughput is to the maximum allowable, the higher the "cost" of that configuration. In contrast, having a significantly lower capacity utilization than is allowable may be good, but being 50% full is not "twice as good" as being 100% full. As a result, the cost is nonlinear, with dramatic increases close to the maximum allowed and jumps to extremely high costs when exceeding the maximum allowed. Finally, there are costs to reassigning clients to new servers. We cover each in turn.

Variable	Description	Scope	Typical values
C_{basic}	weighted sum of skews	system-wide	0-2
C_{fit}	fit penalty	overloaded server	1000's
C_{util}	sum of storage and throughput utilization metrics	server	0-1000's
$C_{movement}$	sum of movement penalties	server	0-10's
D_S, D_T	standard deviation (skew) of {storage, throughput} utilizations	system-wide	0-1
$U_{i,s}$	storage utilization of node *i*	server	0-1
$U_{i,t}$	throughput utilization of node *i*	server	0-1
α	weight for storage skew relative to throughput		0.8
m	number of servers		

Table 1: Components of the cost function and related variables.

Skew

The basic cost starts with a weighted sum of the standard deviations of the capacity and throughput utilizations of the storage appliances:

$$C_{basic} = \alpha D_S + (1-\alpha) D_T,$$

where α is a configurable weight (defaulting to 0.8), D_S is the standard deviation of storage utilizations $U_{i,s}$ (the *s*torage utilization of node *i* is between 0 and 1, or above 1 if node *i* is overloaded), and D_T is the standard deviation of *t*hroughput utilizations $U_{i,t}$ (similar definition and range). The notion is that if predicted utilization is completely equal, there is no benefit to adjusting assignments and increasing that skew; however, one might redefine this metric to exclude one or more systems explicitly targeted to have excess capacity for future growth. Since throughput is more dynamic than capacity, the default weights emphasize capacity skew (80%) over throughput skew (20%).

Overflowing Clients

There are then some add-ons to the cost to account for penalties. First and foremost, if a server would be unable to fit all the clients assigned to it, there is a penalty for each client that does not fit:

$$C_{fit} = \texttt{fit_penalty_factor} \sum_{i=1}^{m} F_i,$$

F_i is the number of clients not fitting on node *i*, and the penalty factor used in our experiments is a value of 1000 per excess host. We use 1000 as a means of ensuring a step function: even if one out of many servers is just minimally beyond its storage capacity (i.e., a utilization of 1.00...01) the cost will jump to 1000+. In addition, when several clients do not fit, the contribution to total cost from the fit penalty is in the same range as the contribution from the utilization (see below).

To count excess clients, we choose to remove from smallest to largest until capacity is not exceeded: this ensures that the greatest amount of storage is still allocated, but it does have the property that we could penalize many small clients rather than removing a single large one. We consider the alternate approach, removing the largest client first, in §7.2.

Utilization

There are also level-based costs. There are two thresholds, an *upper* threshold above (100%), a clearly unacceptable state, and a *lower* threshold (80% of the maximum capacity or throughput) that indicates a warning zone. The costs are marginal, similar to the U.S. tax system, with a very low cost for values below the lower threshold, a moderate cost for values between the lower and upper thresholds, and a high cost for values above the upper threshold, the overload region. Since the costs are marginal, the penalty for a value just above a threshold is only somewhat higher than a value just below that threshold, but then the increase in the penalty grows more quickly with higher values.

The equation for computing the utilization cost of a configuration is as follows. Constant scaling factors ranging from 10–10000 are used to separate the regions of bad configurations: all are bad, but some are worse than others and get a correspondingly higher penalty. The weight of 0.1 for the more lightly loaded utilization makes adjustments in the range of the other penalties such as utilization skew. Each range of values inherits from the lower ranges; for example, if $U_{i,s} > 1$ then its penalty is 10,000 for everything above the threshold of

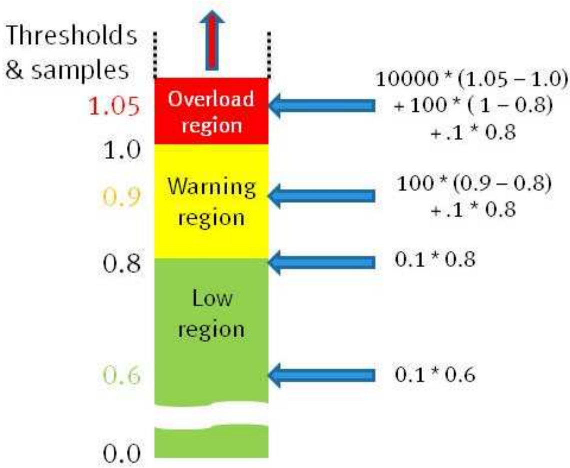


Figure 2: The cost associated with storage on a node, S_i, depends on whether the utilization falls into the *low* region, the *warning* region, or the **overload** region. The costs are cumulative from one region to a higher one.

1, added to the penalty for values between 0.8 and 1 (100 * .2, the size of that region) and the penalty for values between 0 and 0.8 (.1 * .8, the size of *that* region). Figure 2 provides an example with several possible utilization values within and between the regions of interest.

$$C_{util} = \sum_{i=1}^{m} S_i + T_i$$

$$S_i = \begin{cases} .1 * U_{i,s}, & U_{i,s} < .8 \\ .1 * .8 + 100 * (U_{i,s} - .8), & .8 < U_{i,s} \leq 1 \\ .1 * .8 + 100 * .2 + 10000 * (U_{i,s} - 1), & U_{i,s} > 1 \end{cases}$$

$$T_i = \begin{cases} 0, & U_{i,t} < .8 \\ 10 * (U_{i,t} - .8), & .8 < U_{i,t} \leq 1 \\ 10 * .2 + 1000 * (U_{i,t} - 1), & U_{i,t} > 1 \end{cases}$$

The highest penalty is for being $>$ 100% storage capacity, followed by being $>$ 100% throughput. If an appliance is above the lower threshold for capacity or throughput, a lesser penalty is assessed. If it is below the lower threshold, no penalty is assessed for throughput, and a small cost is applied for capacity to reflect the benefit of additional free space. (Generally, a decrease on one appliance is accompanied by an increase on another and these costs balance out across configurations, but content overlap can cause unequal changes.) These penalties are weights that vary by one or more orders of magnitude, with the effect that any time one or more storage appliances is overloaded, the penalty for that overload dominates the less important factors. Only if no appliance has capacity or throughput utilization significantly over the lower threshold do the other penalties such as skew, data movement, and small differences in utilization, come into play.

Within a given cost region, variations in load still provide an ordering: for instance, if a server is at 110% of its capacity and a change in assignments brings it to 105%, it is still severely loaded but the cost metric is reduced. As a result, that change to the configuration might be accepted and further improved upon to bring utilization below 100% and, hopefully, below 80%. Dropping capacity below 100% and avoiding the per-client penalties for the clients that cannot be satisfied is a big win; this could result in shifting a single large client to an overloaded server in order to fit many smaller ones. Conversely, the reason for the high penalty for each client that does not fit is to ensure that the cost encompasses not only the magnitude of the capacity gap but also the number of clients affected, but there is a strong correlation between C_{fit} and C_{util} in cases of high overload.

Movement

The final cost is for data movement: if a client was previously assigned to one system and moves to another, a penalty is assessed in proportion to that client's share of the original system's capacity. This penalty is weighted by a configurable "movement penalty factor." Thus, a client with 1TB of post-dedupe storage, moving from a 30-TB server, would add $movement_penalty_factor * \frac{1}{30}$ to the configuration cost.

$$M_i = \sum_{\texttt{clients}_i} \texttt{movement_penalty_factor} * \frac{\texttt{size}_{client}}{\texttt{size}_i}$$

$$C_{movement} = \sum_{i=1}^{m} M_i$$

Movement_penalty_factor defaults to 1, which also results in the adjustment to the cost being in the same range as skew, though the *movement_penalty_factor* could be higher in a WAN situation. We discuss other values below.

In total, the cost C for a given configuration is:

$$C = C_{basic} + C_{fit} + C_{util} + C_{movement}$$

The most important consideration in evaluating a cost is whether it indicates overload or not; among those with low enough load, any configuration is probably acceptable. In particular, penalties for movement are inherently lower than penalties for overload conditions, and then among the non-overloaded configurations, any with movement is probably worse than any that avoids such movement. Thus the weight for the movement penalty, if at least 1 and not orders of magnitude higher, has little effect on the configuration selected.

4.2 Algorithmic Approaches

We considered four methods of assigning clients; three are fairly simple and efficient but tend to work badly in overloaded environments, while the fourth is much more computationally expensive but can have significant benefits. In all cases, if a configuration includes predetermined assignments, those assignments are made first, and then these methods are used to assign the remaining clients. Existing assignments can constrain the possible assignments in a way that makes imbalance and overload likely, if not unavoidable.

The three "simple" algorithms are as follows. None of them takes content overlap into account in *selecting* a server for a particular client. However, they do a limited form of accounting for content overlap once the server is assigned. The next section discusses extensive analysis to compute *pair-wise* overlaps between specific hosts, but computing the effects of the pair-wise overlaps as each client is assigned is expensive. The simple algorithms instead consider only *class-wise* overlaps, in which the presence of another client in the same "class" as the client most recently added is assumed to provide a fixed reduction to the newer client's space requirements. That reduction is applied to that client before continuing, so servers with many overlapping clients can be seen to have additional capacity. The final, more precise, cost calculation is performed after all assignments are completed.

Random (RAND) Randomly assign clients to backup servers. RAND picks from all servers that have available capacity. If a client does not fit on the selected server, it then checks each server sequentially. If it wraps around to the original selection and the client does not fit, the client is assigned to the first choice, whose utilization will now be > 1, and the cost metric will reflect both the high utilization and the overflowing client. By default, we run RAND 10 times and take the best result, which dramatically improves the outcome compared to a single run [14].

Round-robin (RR) Assign clients to servers in order, regardless of the size of the client. Again, if a server does not have sufficient capacity, the next server in order will be tried; if no server is sufficient, the first one will be used and an overflowing client will be recorded.

Bin-packing (BP) Assign based on capacity, in decreasing order of required capacity, to the server with the most available space. If no server has sufficient capacity, the one with the most remaining capacity (or the least overflow) will be selected and the overflowing client will be recorded.

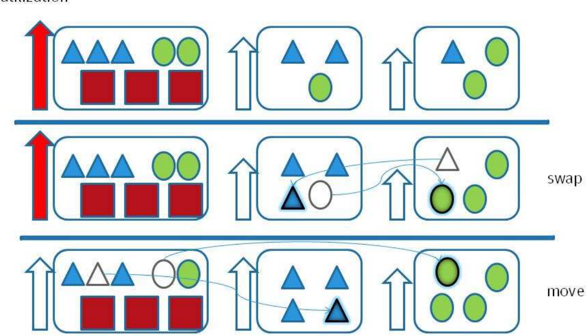


Figure 3: With simulated annealing, the system tries swapping or moving individual clients to improve the overall system cost. Here, the different shapes are assumed to deduplicate well against each other, so swapping a circle with a triangle reduces the load of both machines. Then moving a circle and a triangle from the overloaded server on the left onto the other systems increases their loads but decreases the leftmost server's load. The arrows represent storage utilization, with the red ones highlighting overload. The dark borders and unshaded shapes represent new or removed assignments, respectively.

The fourth algorithm bears additional detail. It is the only one that dynamically reassigns previously assigned clients, trading a movement penalty for the possible benefit of lowered costs in other respects. It does a full cost calculation for each possible assignment, and does many possible assignments, so it is computationally expensive by comparison to the three previous approaches.

Simulated annealing (SA) [11] Starting with the result from BP, perturb the assignments attempting to lower the cost. At each step, a small number of servers are selected, and clients are either *swapped* between two servers or *moved* from one to another (see Figure 3). The probability of movement is higher initially, and over time it becomes more likely to swap clients as a way of reducing the impact. The cost of the new configuration is computed and compared with the cost of the existing configuration; the system moves to the new configuration if it lowers the cost or, with some smaller probability, if the cost does not increase dramatically. The configuration with the lowest cost is always remembered, even if the cost is temporarily increased, and used at the end of the process.

We use a modified version of the Perl `MachineLearning::IntegerAnnealing` library,[1]

[1] This library appears to have been superseded by the AI::SimulatedAnnealing library, `http://search.cpan.org/~bfitch/AI-SimulatedAnnealing-1.02/`.

which allows some control over the way in which the assignments are perturbed:

- The algorithm accepts a set of initial assignments, rather than starting with random assignment.
- It accepts a specification of the percent of assignments to change in a given "trial," when it tries to see if a change results in a better outcome. This percentage, which defaults to 10%, decreases over time.
- The probability of moving a client from one storage appliance to another or swapping it with a client currently assigned to the other appliance is configurable. It starts at $\frac{2}{3}$ and declines over time.
- The choice of the target systems for which to modify assignments can be provided externally. This allows it to focus on targets that are overloaded rather than moving assignments among equally underloaded systems.

By default, SA is the only algorithm that reassigns a client that has already been mapped to a specific storage appliance (we consider a simple alternative to this for the other algorithms in §7.1).

We evaluate the effectiveness of these algorithms in §6.3. In general, RAND and RR work "well enough" if the storage appliances are well provisioned relative to the client workload and the assignments are made on an empty system. However, if we target having each system around 80–90% storage utilization or adjust a system that was overloaded prior to adding capacity, these approaches may result in high skew and potential overload. BP works well in many of the cases, and SA further improves upon BP to a limited extent in a number of cases and to a great extent in a few extreme examples. SA has the greatest benefit when the system is overloaded, especially if the benefits of content overlap are significant, but in some cases it is putting lipstick on a pig: it lowers the cost metric, but the cost is still so high that the difference is not meaningful. Naturally, the solution in such cases is to add capacity.

5 Computing Overlap

There are a number of ways by which one can determine the overlap of content on individual systems. In each case we start with a set of "fingerprints" representing individual elements of deduplication, such as chunks. These fingerprints need not be as large as one would use for actual deduplication. (For instance, a 12-byte fingerprint with a collective false positive rate of $\frac{1}{2^{32}}$ is fine for estimating overlap even if it would be terrible for actually matching chunks – for that one might use 20 bytes or more, with a false positive rate of $\frac{1}{2^{96}}$.) The fingerprints can be collected by reading and chunking the file system, or by looking at existing backups that have already been chunked.

Given fingerprints for each system, we considered two basic approaches to computing overlap: **sort-merge** and **Bloom filters** [1].

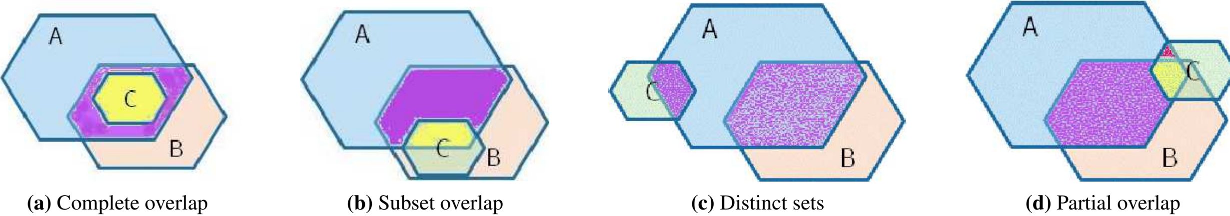
With sort-merge, the fingerprints for each system are sorted, then the minimal fingerprint across all systems is determined. That fingerprint is compared to the minimal fingerprint of all the systems, and a counter is incremented for any systems that share that fingerprint, such that the pair-wise overlap of all pairs of systems is calculated. After that fingerprint is removed from the ordered sets containing it, the process repeats.

With Bloom filters, the systems are processed sequentially. Fingerprints for the first system are inserted into its Bloom Filter. Then for each subsequent system, fingerprints are added to a new Bloom filter, one per system. When these fingerprints are new to that system, they are checked against each of the previous systems, but not added to them.

The sort-merge process can be precise, if all fingerprints are compared. Bloom filters have an inherent error rate, due to false positives when different insertions have collectively set all the bits checked by a later data element. However, that false positive rate can be fairly low (say 0.001%), depending on the size of the Bloom filter and the number of functions used to hash the data.

If the Bloom filters are all sufficiently sparse after all insertions have taken place, another way to estimate overlap is to count the number of intersecting bits that have been set in the bit-vector; however, for "standard-size" Bloom filters setting multiple bits per element inserted, we found it is easy to have a 1% overlap of fingerprints result in 20–30% overlap in bits. Each filter would need to be scaled to be significantly larger than would normally be required for a given number of elements, which would in turn put more demands on system memory, or the number of bits set for each entry would have to be reduced, increasing the rate of false positives. (Consistent with this result, Jain, et al. [10], reported a detailed analysis of the false positive rate of intersecting Bloom filters, finding that it is very accurate when there is high overlap but remarkably misleading in cases of little or no overlap. Since we expect many systems to overlap by 0–20% rather than 75–100%, Bloom filter intersection would not be helpful here.)

Regardless of which approach is used, there is an additional concern with respect to clustering more than two clients together. Our goal is to identify what fraction of a new client *A* already exists on a system containing data



(a) Complete overlap **(b)** Subset overlap **(c)** Distinct sets **(d)** Partial overlap

Figure 4: Four views of possible overlap among A, B, and C. The red or magenta areas indicate overlap that can be attributed to a single pair. The yellow area indicates overlap that must be attributed to multiple intersecting datasets.

from clients B, C, $\ldots Z$. This is equivalent to taking the intersection of A's content with the *union* of the content of the clients already present:

$$Dup(A) = A \cap (B \cup C \cup \ldots \cup Z)$$

However, we cannot store the contents of every client and recompute the union and intersection on the fly. To get an accurate estimate of the intersection, we ideally want to precompute and store enough information to estimate this value for all combinations of clients. If we only compute the number of chunks in common between A and B, A and C, and B and C, then we don't know how many are shared by all of A, B, and C. For example, if $A \cap B = 100$, $A \cap C = 100$, and $B \cap C = 100$, $A \cap B \cap C$ may be 100 as well, or it may be 0. If A and B are already assigned to a server and then C is added to it, C may have as little as 100 in common with the existing server or it may have as many as 200 overlapping. The value of $A \cap B \cap C$ provides that quantity.

Figure 4 depicts some simple scenarios in a three-client example. In the first two cases, $C \subset B$, so even though C overlaps with A the entire overlap can be computed by looking at A and B. In the third case, B and C are completely distinct, and so if A joined a storage appliance with B and C the content in $A \cap B$ and $A \cap C$ would all be duplicates and the new data would consist of the size of A minus the sizes of $A \cap B$ and $A \cap C$. The last case shows the more complicated scenario in which B and C partially intersect, and each intersects A. Here, the yellow region highlights an area where A intersects both B and C, so subtracting $A \cap B$ and $A \cap C$ from A's size would overestimate the benefits of deduplication. The size of the region $A \cap B \cap C$ must be counted only once.

Therefore, the initial counts are stored for the largest group of clients. By counting the number of chunks in common among a set S of clients, we can enumerate the $2^{|S|}$ subsets and add the same number of matches to each subset. Then, for each client C, we can compute the fraction of its chunks that are shared with any set of one or more other clients; this similarity metric then guides the assignment of clients to servers.

To keep the overhead of the subset enumeration from being unreasonable, we cap the maximum value of $|S|$. Fingerprints that belong to $> S_{max}$ clients are shared widely enough not to be interesting from the perspective of content-aware assignment, for a couple of reasons: first, if more clients share content than would be placed on a single storage appliance, the cluster will be broken up regardless of overlap; and second, the more clients sharing content, the greater the odds that the content will exist on many storage appliances regardless of content-aware assignment. Empirically, a good value of S_{max} is in the range $[\frac{S}{3}, \frac{S}{2}]$.

In summary, for each client, we can compute the following information:

- What fraction of its chunks are completely unique to that client, and will not deduplicate against any other client? This value places an upper bound on possible deduplication.

- What fraction of its chunks are shared with at least $S_{max} - 1$ clients? We assume these chunks will deduplicate on any appliance that already stores other clients, providing an approximate lower bound on deduplication, but there is an inherent error from such an assumption: if the $S_{max} - 1$ clients are all on a single appliance, the Sth client will only get the additional deduplication if it is co-resident with these others.

- How much does the client deduplicate against each other client, excluding the common chunks?

Combining the per-pair overlaps with per-triple data, we can identify the best-case client with which to pair a given client for maximum deduplication, then the best-case second client that provides the most *additional* deduplication beyond the first matching client. §6.2 describes the results of this analysis on a set of 21 Linux systems. Since even the 3rd client is usually a marginal improvement beyond the 2nd, we do not use overlap beyond pairwise intersections in our experiments.

5.1 Approximation Techniques

Dealing with millions of fingerprints, or more, is unwieldy. In practice, as long as the fingerprints are uniformly distributed, it is possible to estimate overlap by sampling a subset of fingerprints. This sampling is similar to the approach taken by Dong, et al., when routing groups of chunks based on overlapping content [4], except that the number of chunks in a group was limited to a relatively small number (200 or so). Thus in that work, the quality of the match degraded when sampling fewer than $\frac{1}{8}$ fingerprints, but when characterizing entire multi-GB or multi-TB datasets, we have many more fingerprints to choose from. Empirically, sampling 1 in 1024 fingerprints has proven to be about as effective as using all of them; we discuss this further in §6.2.1.

In addition, it is possible to approximate the effect of larger clusters by pruning the counts of matches whenever the number is small enough. For instance, if $A \cap B$ is 10% of A and 5% of B, $A \cap C$ is 15% of A and 5% of C, and $A \cap B \cap C$ is 0.5% of A, then we estimate from $A \cap B$ and $A \cap C$ that adding A to B and C will duplicate 25% of A's content. This overestimates the duplication by 0.5% of A since it counts that amount twice, but the adjustment is small enough not to affect the outcome. Similarly, in Figure 4d, the yellow region of overlap $A \cap B \cap C$ is much greater than the intersection only between A and C that does not include B: adding A to B and C is approximately the same as adding A to B alone, and C can be ignored if it is co-resident with B.

This approximation does not alleviate the need to compute the overlap in the first place, since it is necessary to do the comparisons in order to determine when overlap is negligible. But the state to track each individual combination of hosts adds up; therefore, it is helpful to compute the full set, then winnow it down to the significant overlaps before evaluating the best way to cluster the hosts. This filter can be applied all the way at the level of pairs of clients, ignoring pairs that have less than some threshold (such as 5%) of the content of at least one client in common.

6 Evaluation

In this section we describe the use of the client assignment tool in real-world and simulated environments. §6.1 discusses the datasets used, §6.2 reports some examples of overlapping content and the impact of sampling the dataset fingerprints, and §6.3 compares the various algorithms introduced in §4.2.

6.1 Datasets

To evaluate our approach, we draw from three datasets:

1. *Linux workstations, full content*. We have a set of 21 collections of fingerprints of content-defined chunks on individual Linux workstations and file servers. Most of these are drawn from a canonical internal test dataset[2] from 2005-6 containing full and incremental backups of workstations over a period of months; since duplicate fingerprints are ignored, this is the union of all content on these systems over that period (excluding any data that never makes it to a backup). About $\frac{1}{4}$ are from a set of workstations and file servers currently in the Princeton EMC office, collected in 2011 through a single pass over each local file system.

2. *Artificial dataset, no content*. In order to show the effect of repeatedly adding clients to a system over time, we generated an artificial dataset with a mix of three client sizes. Each iteration, the system adds 20 clients: 10 `small` clients with full backups of 20GB, 7 `medium` 100GB clients, and 3 `big` 2TB clients. This adds up to 6.9TB of total full backups, which scales to about 8TB of unique data to be retained over a period of several months. We simulate writing the datasets onto a number of DD690 backup systems with 35.3TB storage each; after deduplication, about 5 sets of clients (100 clients in total) can fit on one such appliance. We start with 2 servers and then periodically add capacity: the goal is to go from comfortable capacity to overload, then repeatedly add a server and add more clients until overloaded once again. This can be viewed as an outer loop, in which DD690 appliances are added, and an inner loop, in which 20 clients are assigned per iteration. Once assigned to a server, a client starts with a preference for that server, except for when a new backup server is added: to give the the non-migrating algorithms a chance to rebalance, the previous assignments are forgotten with $\frac{1}{3}$ probability.

   We consider two types of overlap, one in which there is a small set of clients with high overlap, and one in which all clients of a "class" have small overlap. In the former case, each client added during an iteration of the outer loop deduplicates 30% of its content with the corresponding clients from previous iterations of the outer loop: the ith client added when there were 6 DD690s dedupes well with the ith client added when there were [2..5] DD690s present. It deduplicates 10% with all other clients of the same type (big, medium, or small). In the latter case, only the 10% per-class overlap applies.

[2] This is the "workstations" dataset in a previous paper [4].

ID	Best Possible (%)	Best2 (%) =	Widely Shared (%) +	Saved1 (%) +	Add'l Saved (%)	Chunks	Unique Chunks	Match1 host	Match2 host	Pct Saved2 (in isolation)
host1	73.87	73.75	0.77	48.9	24.08	823,256	215,083	host21	host16	30.9
host2	32.06	31.53	0.53	27.8	3.19	9,065,414	6,158,755	host16	host20	3.6
host3	18.68	17.21	0.80	14.9	1.51	3,843,577	3,125,766	host4	host20	12.1
host4	15.03	14.53	0.84	13.2	0.50	4,852,119	4,122,931	host5	host20	10.2
host5	14.34	13.04	0.74	10.5	1.80	6,645,378	5,692,506	host9	host20	7.8
host6	13.00	12.43	2.82	8.9	0.71	7,853,942	6,832,555	host9	host11	1.1
host7	11.67	10.19	5.94	3.3	0.95	3,460,930	3,057,042	host11	host16	2.4
host8	10.88	10.7	8.14	2.5	0.06	2,458,516	2,191,010	host19	host13	1.2
host9	10.3	8.05	0.44	5.7	1.91	31,410,032	28,176,318	host16	host5	2.2
host10	9.41	8.91	5.32	2.7	0.89	4,195,226	3,800,335	host13	host18	2.2
host11	8.73	6.16	2.81	1.9	1.46	8,066,949	7,362,355	host9	host13	1.6
host12	8.36	7.38	4.67	1.6	1.11	4,512,393	4,135,327	host6	host13	1.4
host13	7.89	6.47	3.61	2.1	0.76	6,231,280	5,739,526	host11	host18	1.3
host14	7.88	7.28	5.07	1.9	0.31	4,361,658	4,018,166	host19	host11	1.6
host15	7.70	7.08	4.34	2.4	0.34	5,141,613	4,745,660	host11	host13	1.0
host16	7.36	6.28	0.10	3.9	2.28	64,735,211	59,973,773	host2	host9	2.7
host17	7.17	6.52	2.27	3.3	0.96	3,035,582	2,817,910	host16	host5	1.2
host18	6.27	5.55	2.44	2.2	0.91	9,220,185	8,641,937	host16	host11	1.6
host19	4.73	3.99	2.39	1.2	0.40	9,359,512	8,917,158	host11	host5	0.4
host20	3.07	2.33	0.15	1.8	0.38	28,381,188	27,508,835	host5	host2	1.1
host21	1.77	1.70	0.03	0.9	0.77	43,045,905	42,284,086	host1	host16	0.9
Average	8.13					260,699,866	239,517,034			

Table 2: Inter-host deduplication of 21 workstations

3. *Customer backup metadata, no content*. We have a collection of 480 logs of customer backup metadata, including such data as the size of each full or incremental backup, the duration of each backup, the retention period, and so on. These logs do not include actual content, though they include the "class" that each machine being backed up is in: one can infer better overlap between machines in the same class than machines in different classes, but not quantify the extent of the overlap. (We assume a 10% overlap for clients in the same class.) We preprocess these logs to estimate the requirements for each client within a given customer environment, compute the size and number of backup storage appliances necessary to support these clients, then assign the clients to this set of storage appliances. By adjusting the desired threshold of excess capacity, we can vary the contention for storage capacity and I/O. In this paper we consider only the largest of these customer logs, with nearly 3,000 clients.

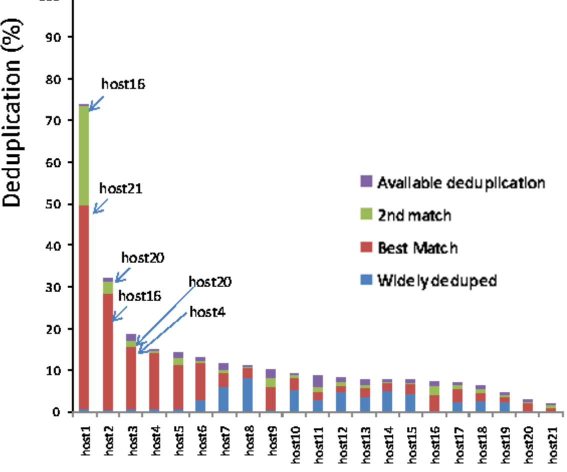


Figure 5: This is a visual depiction of the data in Table 2, showing the components contributions to the deduplication of each host.

6.2 Content Overlap

Table 2 and Figure 5 describe the intersection of the 21 Linux datasets. Hosts are anonymized via the names "host1" to "host21," shown in the first column of the table. The next column shows the idealized deduplication, computed by dividing the number of unique chunks by the number of chunks and subtracting the result from 100%. (We assume that all chunks are the same size, although in practice they are statistically an *average* of

8 Kbytes.) Hosts are sorted by the best possible deduplication rate. The average across all hosts is about 8%.

The next column, **Best2**, reports the deduplication obtained by matching a given host against the best two hosts, as described in §5. It is the sum of the widely shared data appearing on that host (typically around 1–2% but as high as 8%), the additional deduplication specifically against the **Match1 host**, and the additional deduplication against **Match2 host**. Since the second match excludes both common chunks and anything on **Match1**, the added benefit from the second host is usually under 2%, but in the case of `host1`, the second matching host provides about half as much deduplication as the first host, over 24%. **Pct Saved2** indicates how much deduplication could have been achieved by the second host without the first.

The columns listing which hosts provided the best and second-best deduplication indicate that a handful of hosts provide most of the matches. Also, the relationships are not always symmetric, in part because of varying dataset sizes. `Host2` is the best match for `Host16` and vice-versa, but in other cases it is more of a directed graph.

Figure 5 shows this data visually. The height of each bar corresponds to the best possible deduplication, The blue bar at the bottom is the percent of chunks on that host that appear on many other hosts, the red bar shows the additional benefit from the best single match, the green bar shows the additional benefit from a second host, and the purple bar shows extra deduplication that might be obtained through three or more co-resident hosts. Not all bars are visible for each host. For the first three hosts, arrows identify the matching hosts shown in the table. A host with relatively little data may deduplicate well against a larger host, while the larger host gets relatively little benefit from deduplicating in turn against the smaller one; in this case the host with the best overall deduplication matches the host with the poorest deduplication, as a fraction of its total data.

Lest there be a concern that there is a small number of examples reflecting good deduplication, while the average is relatively low, there are other meaningful datasets with substantial overlap. For example, two VMDK files representing different Windows VMware virtual machine images used by an EMC release engineering group overlapped by 49% of the chunks in each.

6.2.1 Sampling

Our goal for sampling is to ensure that even with approximation, the system will find the same "best match" as with perfect data (a.k.a. the "ground truth"), or at least a close approximation to it. We use the following criteria:

- If a host H had a significant match with at least one other host H_1 of 5% of its data, above and beyond the "widely shared" fingerprints, we want the approximated best match to be close to the ground truth. We define "close" as a window β around the correct value, which is within either 5%, 10%, or 20% of the value, with a minimum of 1%. For example, if the ground truth is 50%, acceptable $\beta = 5\%$ approximations would be 47.5–52.5%, but if the ground truth is 5%, values from 4–6% would be acceptable. Note that if the estimated match were outside that range but H_1 was believed to be the best match, we might cluster the two together but misestimate the benefit of the overlap.

- If the best match found via approximation is with another host H_2, rather than the ground truth best match, it may still be acceptable. The approximate overlap needs to be close to the actual overlap of H_2, or we would misestimate the benefits, but we only would find the alternate host H_2 acceptable if it was within β of the value of H_1. Thus the approximate match $H_{2,approx}$ must be $> (1-\beta)H_1$ and $< (1+\beta H_2)$.

- If the host had *no* significant match ($> 5\%$) with another single host, we want the approximation to reflect that. But again, a small change is acceptable. For example, if the best match were 4.5% and we would have ignored it, but the approximation reports that the best match is 5.5%, that is a reasonable variance. If the best match was 1% and is now reported as 5.5%, that would be a significant error.

The ranges of overlap are important because in practice a high relative error is inconsequential if the extent of the match is limited to begin with. If we believe two clients match in only 0.5% of their data, we are unlikely to do much differently if we estimate this match is 1% or 2%, or if we believe there is no match at all. On the other hand, if we think that a 50% match is only 25% or is closer to 100%, the assignment tool might make a bad choice. Even if it picks the right location due to the overlap, it will underestimate or overestimate the impact on available capacity.

Figure 6 depicts the effect of sampling fingerprints, using the same 21-client fingerprint collection. The x-axis depicts the sampling rate, with the left-most point corresponding to the ground truth of analyzing all fingerprints. As the graph moves to the right, the sampling rate is reduced. There are three curves, corresponding to margins of error $\beta = 5\%$, $\beta = 10\%$, and $\beta = 20\%$. The y-axis shows the fraction of clients with an error outside the specified β range. For a moderate margin of error there is little or no error until the sampling rate is lower than $\frac{1}{1024}$, though if one desires a tighter bound on β, the error rate increases quickly.

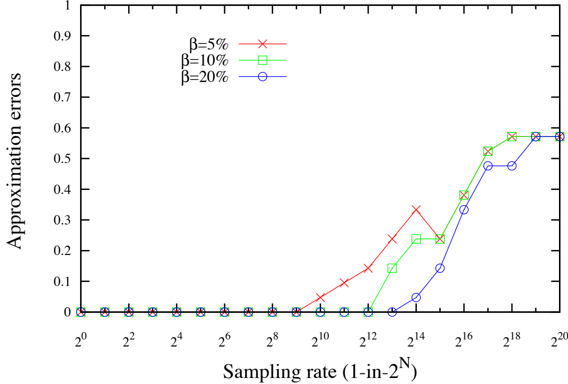


Figure 6: For a given sampling rate, on the x-axis, of $\frac{1}{2^N}$, we compute what fraction of clients have their biggest overlap approximated within an given error threshold β.

6.3 Algorithm Comparison

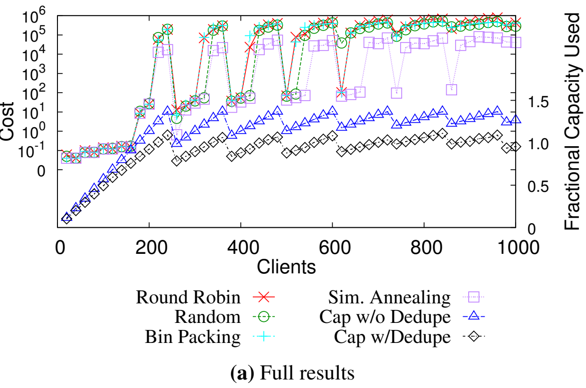
In general, any of the algorithms described in §4.2 work well if the system is not significantly loaded. As capacity or throughput limits are reached, however, the system can accommodate the greatest workloads through intelligent resource allocation. This is especially true if there is significant overlap among specific small subsets of clients.

In our analysis here, we focus on capacity limitations rather than throughput. This is because backup storage appliances are generally scaled to match throughput and capacity, so it is rare to experience throughput bottlenecks without also experiencing capacity shortages. Since it can occur with high-turnover data (a good deal of data being written but then quickly deleted), the cost function does try to optimize for throughput as well as capacity.

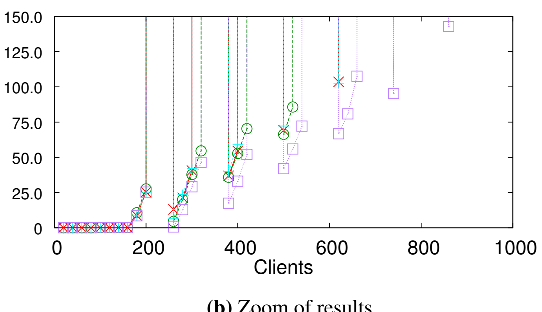
Incremental Assignment

We first compare the four algorithms as clients and backup storage are repeatedly added, using the artificial dataset described in §6.1 and new servers every 120 clients.

Figure 7 shows the results of this process with the number of clients increasing across the horizontal axis and cost shown on the left vertical axis. (Part (a) shows the full range of cost values on a log scale, while (b) zooms in on the values below 150, on a standard scale, to enable one to discern the smaller differences.) The two capacity curves in 7(a) reflect the ratio of the estimated capacity requirements to the available backup storage, with or without considering the effects of the best-case deduplication, and are plotted against the right axis. A value over 1 even with deduplication would indicate a



(a) Full results



(b) Zoom of results

Figure 7: An artificial, homogeneous client population is added 20 hosts at a time, with new backup storage added every 120 hosts after the first 240. A small set of clients match each other with 30% deduplication and otherwise hosts of the same type match 10% of their data. The costs are shown by the curves marked on the left axis. The capacity requirements are shown by the curves at the bottom of the top graph, marked on the right axis.

condition in which insufficient capacity is available, but any values close to or above 1 indicate potential difficulties.

In Figure 7, the general pattern is for the "simple" algorithms to fail to fit all the clients within available constraints, once the collective requirements first exceed available capacity, while SA cycles between being able to accommodate the clients and failing to do so (but still being an order of magnitude lower cost even when failing to fit them). There is a stretch between 600–700 clients in which it does particularly well; this is because in this iteration of the outer loop, the number of distinct clusters of highly overlapping clients equals the number of storage appliances, and the system balances evenly.

While the sequence depicted in Figure 7 is a case in which explicit pair-wise overlap is essential to fitting the clients in available capacity, the sequence in Figure 8 adds fewer clients per storage appliance. Clients almost always fit, though SA improves upon the other approaches some of the time. As expected, RR is not quite

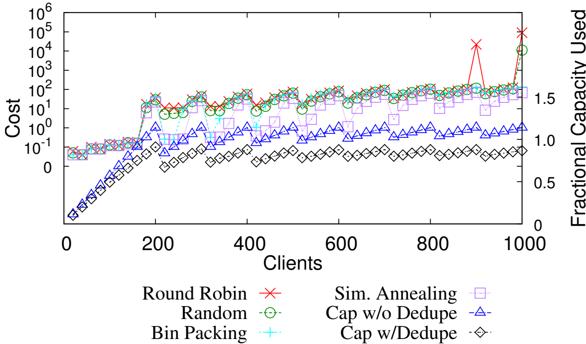


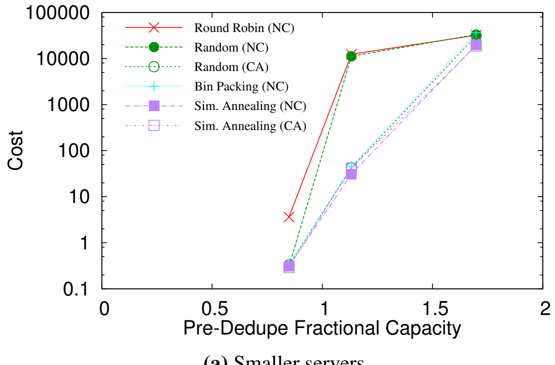
Figure 8: The same artificial, homogeneous client population is added 20 hosts at a time, with new backup storage added every 100 hosts after the first 200. The costs are shown by the higher curves, marked on the left axis. The capacity requirements are shown by the curves at the bottom of the graph, marked on the right axis.

as good as BP; when the number of clients is high, there are cases where RR exceeds capacity because it considers only *whether* a client fits and not how *well* it fits, and because it is constrained by earlier assignments. RAND similarly fails when 1000 clients are present.

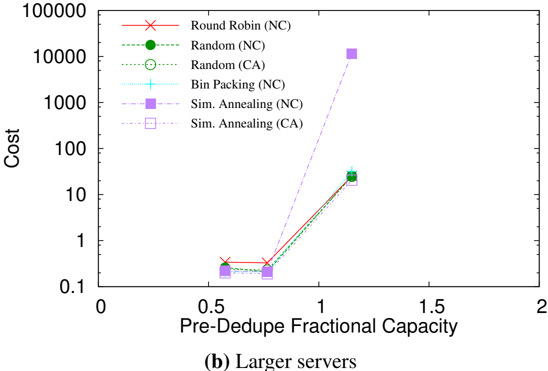
In summary, we find that under high load RAND, RR, and even BP fail to have acceptable costs in a large number of cases, but SA shuffles the assignments to better take advantage of deduplication and fits within available capacity when possible. While the SA results overlap the BP results in some cases, whenever there is a purple square without a matching aqua + overlaid upon it in Figure 7, SA has improved.

Full-Content Client Dataset

Here we describe the effect of assigning the 21-client dataset to a range of backup appliances. The overlaps of the datasets are derived from the full set of fingerprints of each client, but in the case of `Host1`, which is the host that is relatively small but has high overlap, we artificially increase its backup sizes by two orders of magnitude to represent a significant host rather than a trivially small one. Including this change, the clients collectively require 2.92TB before deduplication and 2.46TB or more after deduplication. They are assigned to 2–4 storage appliances with either 0.86TB (“smaller”) or 1.27TB (“larger”) capacity each.[3] For the smaller servers, the clients take from about 70–140% (post-dedupe) of the available storage as the number of backup systems is



(a) Smaller servers



(b) Larger servers

Figure 9: Cost as a function of relative capacity, pre-dedupe, for the modified 21-host dataset, for two backup appliance sizes. Algorithms are either content-aware (CA) or not content-aware (NC).

reduced, corresponding to 85–170% pre-deduplication. For the larger ones, they take 46–92% (deduplicated) or 58–115% (undeduplicated). That is, even with deduplication, at the highest utilization the clients cannot fit on only two of the smaller servers, but they fit acceptably well on the larger ones.

Figure 9 shows the cost as a function of pre-deduplication utilization. For RAND and SA, it presents two variants: one, the content-aware version, is the default; the other selects the lowest cost assuming there is no overlap, then recomputes the cost of the selected configuration with overlap considered. For BP and RR, overlap is considered only to the extent that two clients are in the same class, and the adjustment is made after a given client is assigned to a server (refer to §4.2).

Using smaller servers (9(a)), RR has a slightly higher cost under the lowest load; both RR and RAND (NC) are overloaded under moderate load, and all algorithms are overloaded under the highest load with just two servers. While it is not visible in the figure, SA without factoring content overlap into its decisions is about 6% higher cost than the normal SA which uses that information.

[3] These numbers are taken from early-generation Data Domain appliances and are selected to scale the backup capacity to the offered load. In practice, backup appliances are 1–2 orders of magnitude larger and growing.

Using larger servers (9(b)), the costs across all algorithms are comparable in almost all cases. The notable exception is SA at the highest load: it is overloaded if it ignores content overlap, but fine otherwise. Interestingly, RAND does just as well with or without content overlap, as its best random selection without taking overlap into account proves to be a good selection once overlap is considered.

In other words, at least for this workload, there are times when it is sufficient to load balance blindly, ignoring overlap, and have overlap happen by happy coincidence. But there are times when using overlap information is essential to finding a good assignment: an approach that considers overlap can better take advantage of shared data.

Large Customer Dataset

We ran the assignment tool on the clients extracted from the largest customer backup trace, as described in §6.1. It has nearly 3,000 clients requiring about 325TB of post-deduplicated storage. Using 3 Data Domain DD880s totaling 427TB, these use about 76% of capacity, and all four algorithms assign the clients with a low cost: the maximum is 0.80 for round robin, while BP and SA are 0.24 and 0.23 respectively. It is worth noting that most of the ten RAND runs had costs over 2, but one was around 0.4 and the best was identical to the BP result. SA took over four hours and only improved it from 0.24 to 0.23.

What about overload conditions? If these were just 2 DD880s (285TB), the average storage utilization goes to 114% so no approach can accommodate all the clients. Even so, the cost metric is a whopping 184K for BP, 183K for RR, and 159K for RAND (which, by taking the lower costs of client overlap into account when comparing alternatives is able to find a slightly better assignment). These high costs are dominated by the "fit penalty" due to about 130–160 clients, out of 2983, not fitting on a server. SA, however, brought the cost down to 25K (of which 12K is from 12 clients not fitting). However, it did this by running for 5.5 cpu-days (see the next subsection).

Obviously one would not *actually* try and place 3,000 clients, totaling 325TB of post-dedupe storage, on a pair of 142TB servers. This example is intended to show how the different approaches fair under overload, and it also provides an example of a large-scale test of SA. The large number of clients to choose from poses a challenge, in that a cursory attempt to move or swap assignments may miss great opportunities, but an extensive search adds significant computation time (see the next subsection). Tuning this algorithm to adapt to varying workloads and scales and deciding the best point to prune the search are future work.

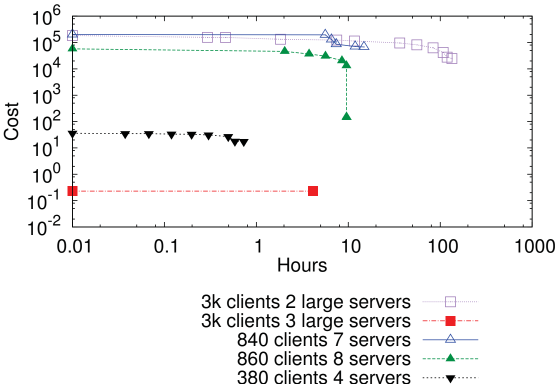


Figure 10: Cost as a function of simulated annealing analysis time for several cases. Both axes use a log scale. Except for the right-most points, any points that appear within a factor of 1.5 in both the x and y values of a point already plotted are suppressed for clarity.

6.4 Resource Usage

While our results have shown that SA can produce better assignments than the other algorithms in certain cases, there is a cost in terms of resource requirements. All three "simple" algorithms are compact and efficient. For example, the unoptimized Perl script running bin-packing on the nearly 3,000 clients and two small servers in the preceding subsection took 163M of memory and ran in 23s on a desktop linux workstation. Running SA on the same configuration took over 5 days, and the complexity of the problem is only increased when pair-wise rather than per-class overlaps are included. For the iterative problem with up to about 1,000 clients and pair-wise overlaps, the script takes several gigabytes of memory and runs for over a half day on a compute server.

Figure 10 shows timing results for five examples of earlier experiments. Two are the large-scale assignments described in the previous subsection, with nearly 3,000 clients that either fit handily or severely overload the servers. The horizontal line at the bottom represents the case where SA runs for over four hours with no effective improvement over a cost that is already extremely low. The curve toward the top with open squares is the same assignment for $\frac{2}{3}$ of the server capacity. SA dramatically reduces the cost, but it is still severely overloaded. The curve (with open triangles) near that one represents one of the incremental assignment cases in which the system is overloaded regardless of SA, while the one just below that has 20 more clients but one additional server and, in the case of SA, has a relatively low cost after a long period of annealing (the sharp drop around the 10-hour mark is an indication of SA finally succeeding in rearranging the assignments to fit capacity). Finally, the re-

maining triangle curve represents a smaller test case in which the cost starts low but SA improves it beyond what BP initially did.

In some cases (not plotted), there is a drop followed by a long tail without improvement. Ideally the process would end after a large score decrease if and only if no substantial decreases are still possible; since there is the potential to miss out on other large improvements, we let SA continue to search and hope for further decreases. Generally, with our default parameters, SA runs for seconds to hours on a desktop computer, but when configuring or updating a backup environment, that is not unreasonable, and the "best solution to date" can be used at any time. The more excess capacity there is, the easier it is for SA to hone in on a good solution quickly. For assignments of thousands of clients in an overloaded environment, some sort of "divide and conquer" approach will be necessary to keep the problem manageable.

7 Variations

In this section we discuss a couple of variations on the policies previously described: "forgetting" assignments and biasing in favor of small clients in the cost function.

7.1 Forgetting Assignments

As described to this point, whenever new clients are added to an existing set of assignments, the first assignments are "carved in stone" for the simple algorithms: they cannot be modified, and only the new unassigned clients can be mapped to any server. The SA algorithm is an exception to this, in that it can perturb existing assignments in exchange for a small movement penalty.

Here we consider a simple but extreme change to this policy: ignore all existing assignments, map the clients to servers using one of the algorithms, and pay movement penalties depending on which clients change assignments. When the assignment that takes previous assignments into account does not cause overflow, starting with a clean slate usually results in a higher cost because the movement penalties are higher than the other low costs from the "good" and "warning" operating regions. But when there would be overflow, it is often the case that rebalancing from start avoids the overflow.

Figure 11 repeats Figure 7(a), with one change: for RR, RAND, and BP, each point is the minimum between the original datapoint and a new run in which the previous assignments were ignored during assignment.[4] Ignoring initial assignments improved the cost metric in 35% of the cases overall, and in 43% of the cases in which the cost was over 1000 (indicating significant overload): it is frequently useful but no panacea.

[4] Due to the high cost of SA, we do not re-run each SA experiment but instead take the minimum of the SA run and the "forgotten" BP run; that is, SA could have started from the lower BP point rather than the previous one that considered previous assignments. It might improve the cost beyond that point, something not reflected in this graph.

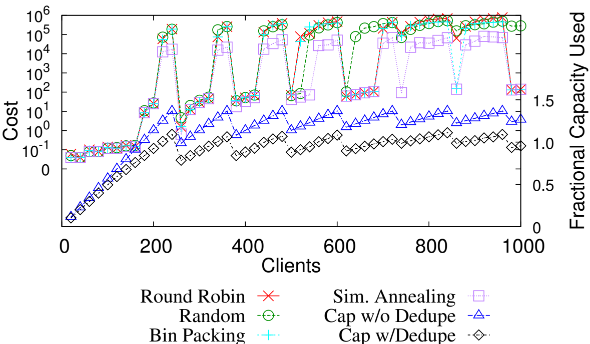


Figure 11: The same clients and servers are assigned as in Figure 7(a), but previous assignments can be ignored in exchange for a movement penalty.

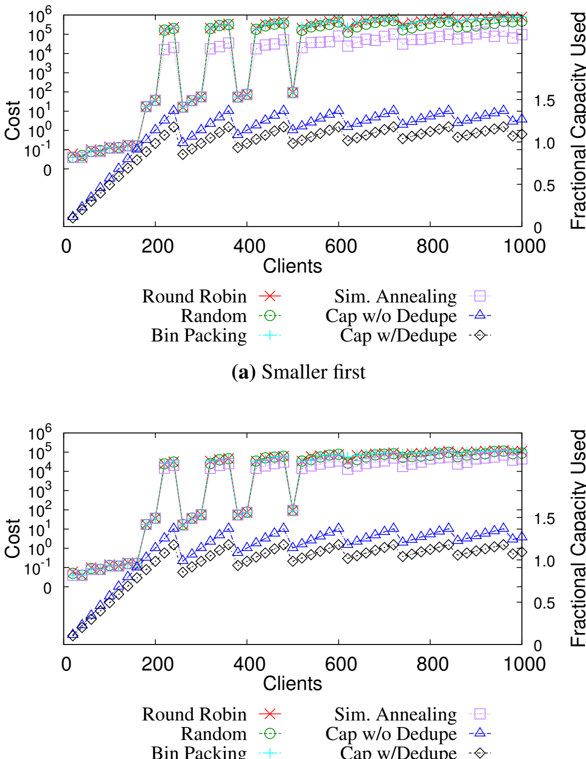
The most notable difference between Figure 7(a) and Figure 11 is in the range of 600–700 clients. Previously we noted that SA does especially well in that range because of overlap, but if BP and RR start there with completely new assignments, they too have a low cost due to keeping better deduplicating clients together.

7.2 Counting Overflow

As described, the cost function biases in favor of large clients: it assumes that it is more important to back up a larger client than a smaller one, so it removes clients in order of size, smallest first, to count the number of clients that do not fit on a server. This approach is intuitive, in that a large client probably *is* more important than a small one, and it also simplifies accounting because if clients are added in decreasing order of size, we can remove a small client without affecting the deduplication of a larger one that remains.

An alternative cost function would minimize the number of occurrences of overflow by removing the largest client(s) to see if what remains will fit. This has the effect of minimizing the extra per-client penalty while still penalizing for exceeding the capacity threshold. In essence, it encourages filling N-1 servers to just below 100% utilization, then placing all the remaining (large) clients on the Nth server.

Figure 12 compares the smallest-first and biggest-first penalties for the example used in Figure 7, modified to exclude the pair-wise 30% deduplication of specific combinations of clients. (This is because recomputing the impact of removing a client against which other clients have deduplicated would require a full re-evaluation of



(a) Smaller first

(b) Bigger first

Figure 12: The same clients and servers are assigned as in Figure 7, but deduplication is only considered within a class (big, medium, or small) rather than having greater deduplication for specific pairs. C_{fit} is computed by removing the (a) smallest or (b) largest clients first.

the cost function, compared with class-wise deduplication, and has not been implemented.) The two graphs look quite similar, but because of the change to the value of C_{fit} the peak values are about one order of magnitude lower when the largest clients are counted. There is no qualitative difference in this example beyond a narrowed gap between SA and the other approaches.

8 Discussion and Future Work

Assigning backups from clients to deduplicated storage differs from historical approaches involving tape because of the stickiness of repeated content on the same server and the ability to leverage content overlap between clients to further improve deduplication. We have accounted for this overlap in a cost function that attempts to balance capacity and throughput requirements and have presented and compared several techniques for assigning clients to backup storage appliances.

When a backup system has plenty of resources for the clients, any assignment technique can work well, and there is little difference between RAND and our most advanced technique with SA. The more interesting case is when capacity requirements reach beyond 80% of what is allocated. We have found that RAND and RR tend to degrade rapidly, while bin-packing and SA continue to maintain a low cost until capacity becomes oversubscribed. In cases of significant overlap, SA is able to use client overlap to increase the effective capacity of a set of deduplicating backup servers, deferring the point at which the system is overloaded.

There are a number of open issues we would like to address:

- evaluation of overlap in a wider range of backup workloads
- evaluation of overlap beyond the “best match” for those cases where cumulative deduplication beyond one other host is significant
- full integration between client assignment and backup software
- use of the assignment tool to manage transient bursts in load due to server failures or changes in workload
- additional evaluation of the various weights and cost function
- optimization of the SA algorithm for large-scale environments; and
- additional differentiation of clients and servers, for instance to route backups to different types of devices automatically depending on their update patterns and deduplication rates.

Efforts to integrate content affinity with pre-sales sizing are already underway.

Acknowledgments

We thank Windsor Hsu, Stephen Manley, Hugo Patterson, Hyong Shim, Grant Wallace, and Jason Warlikowski for helpful comments on the design of the system and on earlier drafts. Mike Mahler, Jason Warlikowski, Yuri Zagrebin, and Jie Zhong provided assistance with the development of the assignment tool. Thanks to Thomas Waung for backup traces and to Benjamin Fitch for the `MachineLearning::IntegerAnnealing` library. We are especially grateful to the anonymous referees and our shepherd, Doug Hughes, for their feedback and guidance.

References

[1] Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (Jul 1970)

[2] Chamness, M.: Capacity forecasting in a backup storage environment. In: LISA'11: Proceedings of the 25th Large Installation System Administration Conference (Dec 2011)

[3] Chapin, S.J.: Distributed and multiprocessor scheduling. ACM Comput. Surv. 28 (March 1996), http://doi.acm.org/10.1145/234313.234410

[4] Dong, W., et al.: Tradeoffs in scalable data routing for deduplication clusters. In: Proceedings of the 9th USENIX conference on File and stroage technologies. FAST'11, USENIX Association (February 2011)

[5] EMC Corporation: Data Domain Boost Software (2010), http://www.datadomain.com/products/dd-boost.html

[6] EMC Corporation: Unified backup and recovery with EMC NetWorker (Feb 2010), http://www.emc.com/collateral/software/white-papers/h3399_nw_bu_rec_wp.%pdf

[7] Gmach, D., Rolia, J., Cherkasova, L., Kemper, A.: Capacity management and demand prediction for next generation data centers. IEEE International Conference on Web Services (2007)

[8] Harchol-Balter, M., Downey, A.B.: Exploiting process lifetime distributions for dynamic load balancing. ACM Trans. Comput. Syst. 15, 253–285 (August 1997), http://doi.acm.org/10.1145/263326.263344

[9] IBM Corporation: Tivoli Storage Manager (2011), http://www-01.ibm.com/software/tivoli/products/storage-mgr/

[10] Jain, N., Dahlin, M., Tewari, R.: Taper: tiered approach for eliminating redundancy in replica synchronization. In: FAST '05: Proceedings of the 4th USENIX Conference on File and Storage Technologies. pp. 21–21 (2005)

[11] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983), http://www.sciencemag.org/content/220/4598/671.abstract

[12] Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. In: Proceedings of the 9th USENIX conference on File and stroage technologies. FAST'11, USENIX Association (February 2011)

[13] Milojicic, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Comput. Surv. 32, 241–299 (September 2000), http://doi.acm.org/10.1145/367701.367728

[14] Mitzenmacher, M.: The power of two choices in randomized load balancing. IEEE Trans. Parallel Distrib. Syst. 12(10), 1094–1104 (2001)

[15] Quinlan, S., Dorward, S.: Venti: a new approach to archival storage. In: FAST '02: Proceedings of the 1st USENIX conference on File and Storage Technologies (2002)

[16] da Silva, J., Gudmundsson, O., Mosse, D.: Performance of a parallel network backup manager. In: USENIX (ed.) Proceedings of the Summer 1992 USENIX Conference: June 8–12, 1992, San Antonio, Texas, USA. pp. 217–226. USENIX (Summer 1992)

[17] da Silva, J., Gumundsson, O.: The Amanda network backup manager. In: USENIX (ed.) Proceedings of the Seventh Systems Administration Conference (LISA VII): November 1–5, 1993, Monterey, CA, USA. pp. 171–182. USENIX (Nov 1993)

[18] Soundararajan, V., Govil, K.: Challenges in building scalable virtualized datacenter management. SIGOPS Oper. Syst. Rev. 44, 95–102 (December 2010)

[19] Symantec Corporation: Next generation data protection with Symantec NetBackup 7 (2011), http://eval.symantec.com/mktginfo/enterprise/white_papers/b-next_genera%tion_data_protection_with_sym_nbu7_WP_20999878.en-us.pdf

[20] Zhang, X., Du, D., Hughes, J., Kavuri, R.: Hptfs: A high performance tape file system. In: Proceedings of 14th NASA Goddard/23rd IEEE conference on Mass Storage System and Technologies (2006)

[21] Zhu, B., Li, K., Patterson, H.: Avoiding the disk bottleneck in the Data Domain deduplication file system. In: FAST '08: Proceedings of the 6th Conference on File and Storage Technologies. pp. 269–282 (February 2008)

Getting to Elastic: Adapting a Legacy Vertical Application Environment for Scalability

Eric Shamow - Puppet Labs

ABSTRACT

During my time in the field prior to joining Puppet Labs, I experienced several scenarios where I was asked to be prepared for so-called "elastic" operations, which would dynamically scale according to end-user demand. This demand only intensified as the notion of moving to IaaS became realistic. There's no button you hit marked "make elastic" to turn your infrastructure into an elastic cloud...rather you need to come to an understanding both of the technologies your organization uses, its tolerances for latency and downtime, as well as your platform, to get there. This paper discusses the key areas that must be addressed: organizational culture, technical policy development, and infrastructure readiness.

Introduction

As I've moved through the industry, it's become increasingly common to find organizations operating what might be termed an "internal cloud" - a commodity hardware infrastructure front-ended by VMware, Xen, or another virtualization technology, being used to cushion the need for rapid and varying server deployments. Over the past few years, I have seen increasing interest in outsourcing that operation - in moving to external cloud offerings including IaaS. In most cases, I've also needed to become prepared for elastic expansion of our apps as we modify them to scale out rather than up.

I encountered many of these problems during the time I spent as Manager of the Systems Operations group at Advance Internet. Advance is a mid-size company in the publishing field, running approximately 1050 servers in a local, private cloud. Although I left Advance prior to the full implementation of our elastic solution, I was deeply involved in the architecture and implementation of that solution, and was fortunate to learn valuable lessons about how to take an entrenched static environment into a dynamic one.

There's no button you hit marked "make elastic" to turn your infrastructure into an elastic cloud...rather you need to come to an understanding both of the technologies your organization uses, its tolerances for latency and downtime, as well as your platform, to get there. Advance traveled some of this road, and this report will include both information about the solutions we found, and some recommendations for those attempting to do the same.

Characterizing the Problem

In order to consider what will be necessary to "go elastic," we must first evaluate what that phrasing really means. How elastic do we want to be? What parts of our applications are able to scale easily? What parts do not? What elements of our process or infrastructure make automatic expansion impossible? In short, what do we need to know?

At Advance, in examining our environment, I identified five major questions or issues that would be show-stoppers for us implementing any kind of scalable environment:

1) Our servers and applications could not be deployed without human intervention. Documentation was limited and there was no automation available.
2) We had no information available about when to deploy a new server automatically. There was a mandate to be able to expand dynamically, but no information about what that meant.
3) Similarly, we did not know when to automatically retire a new server. How responsive to increases and decreases in load would we need to be?
4) What was to be the mechanism for the automatic deployment and retirement?
5) Were our applications optimized to take advantage of this type of scaling? In several cases our experience was that performance improvement was not a linear correlation with an increase in server count - and in fact that in some cases increasing parallelism was damaging to performance. We would need to determine which applications would need to be refactored to handle this architecture, and which were prepared to handle it natively.

In any environment facing similar issues, the five listed above will form the core of the matter - the remainder of our internal fact-finding extended naturally from the answers we found and the process we underwent in attempting to determine those answers.

For those undergoing the same exploration, this fact-finding exercise will form the groundwork for all future work in this space. This means that truthful responses and openness are absolutely necessary. The teams involved don't need to agree on a solution yet, but without a common understanding of the problem space, we cannot reasonably determine whose concerns or enthusiasm are justifiable. It can often help to present this as an opportunity to air long-unaddressed concerns in a new way. If the application team distrusts elasticity, encourage them to fully explain and justify those concerns and promise that they will be addressed as part of the proposed solution. Getting everyone to cooperate here is the most critical step of the process. For me, getting to elastic meant a lot less engineering than I expected, and a whole lot more PR, meetings, and assuaging of concerns.

Elasticity Means Automation

The first key recognition about elastic expansion is that by definition, it means that the server provisioning process must be automated. This is a bridge that many organizations have yet to cross. In some cases the deployment process itself may be automated, but post-install configuration is not completed automatically. My own findings gathered from the organizations I have observed - and this was the case at Advance as much at others - are that most installation and configuration procedures are not automated because groups do not have clear and stable procedures that are followed for deployment. Whether this is because deployment teams do not maintain regular standards for system configuration, or because development teams do not provide accurate release notes or cleanly packaged applications ultimately comes down to finger-pointing; the organization as a whole must recognize that if it wants elasticity, it will need automation, and automation requires clarity of purpose and requirements, and stability of procedures.

Automation itself has multiple components, and depending on the breakdown of roles and responsibilities within an organization, these components are often managed by different groups. Infrastructure groups will have concerns about provisioning storage and network; OS groups will worry about package repositories, OS versioning, and

configuration management; application groups will focus on updating application-specific configurations to recognize new or removed members of a cluster, reshuffling data that has been partitioned based on previous cluster size, and changing various application settings to properly tune performance. All of these are critical and should be clearly mapped.

Where possible, inquiry into how they affect each other is worth discussion - does repartitioning our data suggest different OS configs? With the new cluster size, should we alter our load balancer configuration? However, don't let these advanced discussions derail the primary goal of understanding how your systems are provisioned. The second-level analysis of how those systems interact will occur naturally during the design and implementation of your process, and should continue to iterate through its lifecycle. The most important thing is to come to an understanding of those manual processes which are not currently automated. Those manual steps are your hard roadblocks on the way to elasticity.

Ultimately, at Advance, we settled on a toolset of Kickstart for OS deployments, managed through Cobbler for the additional repository and profile information it permitted. We then handed off to Puppet for application installation and configuration, having worked closely with the application teams to build Puppet manifests that handled their applications appropriately. On the infrastructure side, the SAN, network and VMware team decided to manually script their deployment, resulting in a tool called vDeploy. I will discuss this tool later on in the paper. Ultimately, the tools you choose should be based on two factors: your own comfortability with them, and their flexibility to work well together and to integrate with each other. It is not always critical to choose the best-of-breed software, but rather to choose the software that best fits you and your organization.

Elasticity Requires Open Metrics

An additional component to expanding and contracting an environment in an automated fashion is that accurate and relevant metrics about that environment must be available. In order for those metrics to be meaningful for elasticity, they must be reliable and comprehensive enough that an unattended system can make bottom-line decisions based on them: should I deploy or remove a live system from my customer-facing site immediately? This means that the metrics cannot be siloed as many IT reporting infrastructures are, but must reflect both the state of the application infrastructure as well as the applications running on it. These metrics must also be reliable: they must not be inaccurate, fudged, or intermittently available because of an individual group's desire to hide information from the rest of the team. Elastic expansions and contractions affect the whole without human intervention, but by definition this process is naive - it can only know what we tell it. If we lie to the system, the system will make poor choices.

The choice of metrics should also reflect a cross-disciplinary approach. Much is lost in IT monitoring because of a lack of communication between groups. A monitoring team will pride itself on implementing trend lines for disk utilization, but will fail to monitor a change in a transaction rate or size easily exposed by the monitored application itself. These metrics can predict an increase in the rate of growth at a time when the change would only appear to be a statistical anomaly in the storage data. Again, the discussions of these interrelationships will evolve from the discussions and implementations you are implementing here,

and we shouldn't hesitate too long attempting to nail them down early. That said, any understanding we can get about interrelationships between the components in our environment helps us better predict future changes. Better prediction means better automation, which means elasticity that's less likely to break.

At Advance this was a major source of contention. Monitoring was highly siloed, with Systems controlling an array of Cacti, PNP, MRTG, and proprietary VMware, 3par, and NetApp applications to monitor and graph data - in fact, even within systems, monitoring was siloed, split between different implementations in the DBA, infrastructure, and operations spaces. Application development staff often maintained off-the-radar monitoring systems stashed on workstations or quasi-production servers. The metrics from these groups were never aggregated, and much time was lost bouncing requests and information back between multiple people who were hesitant to allow access to - or knowledge of the existence of - their proprietary systems.

Openness Requires Culture Change

If the organization preparing to implement a model based on elastic expansion is not in the state needed to gather the information above - with a clear availability of infrastructure, OS, and application-level metrics across the board, honest communication between groups and well-documented deployment and configuration changes, elastic expansion is unlikely to be possible. These steps are all pre-requisites for technological change, but they themselves are less technological than cultural. If organizations are going to be prepared for elasticity - operating at a minimum cost most of the time but prepared for the huge onrush of traffic caused by an article "going viral" or the sudden success of their service[1], they must address the underlying lack of transparency before they can begin to work on the technical challenges.

In reality, getting this to happen is often the hardest part of the process. It is fortunate if the change is being implemented in a top-down manner, in that if management is mandating the change, it is often willing to enforce that mandate by requiring teams to cooperate. But what if the change isn't mandated?

In my own experience, the best approach is two-pronged. The first prong is to establish the missing communication. As the head of an Operations team, I regularly met with the head of Development teams, including those of small development groups that my predecessors had often ignored. I wanted to know their pain points, where Operations was letting them down or frustrating their work. Establishing this communication was key to establishing trust.

Trust, however, does not come through words but through deeds. The best action I found I could take in this regard was to surrender unilaterally. I might not be able to get developers or infrastructure to share everything with me, but I would share everything with them. Every incident was clearly documented, metrics were available to all teams, and we developed a process for requesting the addition of new metrics. I committed to making these newly-requested metrics available to them with an response time based on severity, reaching from 20-30 minutes during a crisis, to a maximum of 48 hours outside of one.

I also worked hard to develop a professional chain of command-based communication system with development managers. This may not be applicable in all engineering environments - in many having all

[1] http://blog.pinboard.in/2011/03/anatomy_of_a_crushing/

discussions on a public list is part of the fabric of their work culture. But it can also result in decisions made based on ego and pride rather than technical judgment. Being called out on an error or disagreement in public forces a different type of response from a concern brought quietly in private. At Advance I committed to bring development concerns to the relevant managers and help triage my team's issues rather than exposing them on our internal IRC channels and mailing lists, and asked the development managers to do the same. The ratcheting-down of public tensions combined with the daily give-and-take of triaging priorities with the other managers aided greatly in establishing an understanding of other teams' needs and willingness to cooperate.

Getting Things Started

We've now established communication between departments, established some baseline metrics that we need to pay attention to, and defined clearly the expectation that server rollouts and retirements - from the bare metal phase to appearing in a user-facing cluster - should be automated. Now we're ready to do some work. But where to begin work?

For the purposes of this paper, I will assume that metric collection systems are already available to you, and that you need only tune your existing system to provide you the agreed-upon information. There are a variety of tools excellent at collecting and displaying raw data - from the simplicity of MRTG to more complex tools such as Munin or Cacti, and newer distributed tools such as Graphite or Ganglia. The use of one or more of these will depend on your data sources and the familiarity of your teams with the tools in question. My team used a mix of Cacti and PNP4Nagios, although we were strongly looking into Graphite as a replacement.

Finding Meaningful Metrics

Assuming that we have monitoring technology in place, the next obvious question is "what do we measure?" The answer to this question may at first seem obvious to stakeholders on all sides of the discussion, but a quick synchronization of expectations often indicates that each group's answer is different. The infrastructure and OS groups will tend to monitor metrics focused on the performance of the system itself such as processor load, memory availability, I/O throughput, CPU percentage (distinct from load, which really measures queue length - a distinction lost on many involved in resource monitoring)[2], and swap usage.

In the meantime, the application team will likely be focusing on internal data points that reflect the actual capacity of the application itself, identifying performance of key areas of code, headroom left in caching applications such as Memcache or Varnish, and other data points that reflect how pieces of the code are relating to each other. If there is a separate business owner with access to a dashboard or metrics, that person or group is likely examining more vanilla performance stats - for a web application, time for first byte download, hits per second, and so forth.

It is very likely that none of these metrics will give you on its own the answer that indicates at what point your application will need to elastically expand. In fact, it is likely that, until this point, any discussions about non-elastic expansion have involved meetings between several stakeholders to review this data and find ways to optimize on existing hardware.

---

[2] http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages

Finding the right formula is an exercise in looking at aggregate data patterns, finding correlations that seem to reliably suggest the need for additional servers, and then regularly re-examining those metrics as the application and hardware profiles change.

The worst mistake you can make at this point is assuming that you know or understand too much about your application or environment. What was true several months ago may not be true now...a feature in the application that caused an I/O bottleneck six weeks ago may have been rectified in the application code four weeks ago, and now you've hit a CPU limit on your storage device. Assumptions about causes are more likely to cause bad interpretation of data, which in turn are more likely to cause a misunderstanding of what criteria will need to be used for automated scaling. So the important part of this stage is to have a fresh discussion about application performance and possible bottlenecks at all levels - an informed discussion, but one that makes no assumptions and thoroughly re-examines every facet of the environment looking for hidden indicators and bottlenecks. You won't find them all, but application, operations and business people together will find a lot more than any of those three alone. This is what DevOps looks like in practice.

The Shifting Landscape

Before moving on to the next stage of enabling the elastic environment, I want to return to a phrase used just a few paragraphs back: "What was true several months ago may not be true now."

While this will always be the case is fast-moving, multifaceted IT environments, what should not be the case is that any of this changing truth should be undocumented, or worse a complete surprise to all but one or two people. In a field with rampant hyper-specialization with limited training budgets and one or two "experts" in a given technology per group, it is almost inevitable that sub-pockets of activity have developed which are at least partially invisible, even to members of that pocket's own team.

This type of change is absolutely toxic to elastic expansion. Since all of the painstaking research and rule development you are doing is based around a shared understanding of the environment, changes to that environment that are not automated make it impossible to deploy a single additional node without manual intervention. For that reason, change management must be implemented for an elastic environment to succeed.

This may sound like a leap, but if you examine the nature of elasticity, the reasoning becomes clear. Elasticity is essentially a set of rules wrapped around automation -- a set of conditions under which automated procedures should take place. Automation itself is really nothing more than a form of machine-parseable and actionable documentation - we are taking yesterday's run book or wiki doc and turning it into a YAML file, but in the end we are writing documentation about how a system should be configured, and then using an application to verify compliance with that document.

Note that I did not say that change control was needed - merely change management[3]. As long as the changes made are compatible with the rest of the operating environment and do not interfere with its operation, those changes can be submitted without review. Whether it is wise to do so is a different matter, but don't attempt to bite off more than you can chew here - the framework for change management can be

[3] http://www.technologyexecutivesclub.com/Articles/management/artChangeControl.php

expanded to include change control later on. For now, the important thing is that any change that would affect the ability to automatically rebuild a system is made part of the server and app deployment processes.

Policy

Even when using clearly defined metrics to signal the need for expansion, there are additional factors to consider. First, we must consider the statistical anomaly. If you are running a website, you don't want to scale to a thousand machines because a web crawler hit your site and began to index, or because a user wrote a bad script to fetch your page every millisecond. Similarly, we must consider how long it takes for a new server to come up. Depending on the nature of your environment, this can be very tricky. If load increases sharply, you may need a new server in under a minute. Even with well-automated deployment, a large database server can take five or more minutes to power up and build. If this is not fast enough to save your application from falling over, we have missed the point of the elastic expansion.

The reverse is also true. If load drops to nothing because of an ISP failure, we do not want our production cloud to shrink to its minimum size. We also don't want to power down servers we think we may need again in a few seconds or minutes.

It is the rules around making these determinations that I refer to as "policy" - not a formal organizational policy, but rather an internal technical policy explaining when and how fast you expand, when and how fast you contract, what the artificial limits to both of the above operations should be, and how we work around the elements of those operations that don't fit our environment.

There is no formula that can be generated for this outside of an examination of your own application's behavior and the metrics you should now be gathering. As an example, however, I can discuss the type of solutions we had envisioned at Advance.

For the particular example of database servers, we looked at a combination of server load, database server queue length, and slow query information from the database server, and latency and queue information from the application side, to determine that a new database server was needed. However a new database server could take in excess of ten minutes to provision, far too long to resolve a sudden explosion of activity.

Advance's solution was to mandate that, depending on cluster size, one to two database servers would be provisioned and immediately powered down as par[4]t of our cluster at minimum size. Every time new database servers were automatically provisioned, 1-2 extra servers would be provisioned and immediately powered off. When the need came for new servers, we could begin provisioning additional servers but simultaneously power up the 1-2 idle servers, providing relief to the application within a minute, while additional resources came on line. We employed this strategy in reverse while shutting systems down, decommissioning them but always leaving 1-2 systems powered down but not destroyed.

We also decided to implement several caps on growth and decommissioning to hedge against the possibility of failures in our metrics and formulas. We only allowed growth to proceed at a limited rate, controlling the maximum number of servers that could be provisioned per 15-minute period, and setting a maximum limit on the number of machines that could be auto-deployed without administrator

[4] http://pulpproject.org/

intervention. We set similar limits on decommissioning.

This strategy works well for a “naive” application, where application servers are not aware of each other and can scale out horizontally. This is not the case for most applications, particularly in-house ones which have been written to scale vertically - requiring more resources such as RAM and CPU - rather than horizontally. As a result, many of these apps will not see a linear improvement as each server is added, and it is possible to see a diminishing return, and eventually even a negative impact from the addition of more servers. While an application rewrite down the line should help this, it’s almost never immediately possible; rather, you should tailor your expansion policies to fit the characteristics of the application you have, while encouraging your development teams to begin thinking in terms of horizontal rather than vertical resource usage in the future.

There is an additional concern - application servers which must remain aware of each other - which we will return to after a discussion of the necessary remaining components of the elastic toolset.

Getting the Infrastructure Ready

For the purposes of this discussion, I will assume that the reader is functioning in a “cloud”-type virtualized environment. It is possible to scale elastically in a hardware environment, but the complexity level is much higher. While implementing this system, I was working with an internal cloud built on VMware vSphere, with Infoblox providing DNS and DHCP and Cobbler for provisioning and repository management.

The key infrastructure elements needed to support this are as follows:

- Network support - your network devices must support servers being brought up in a variety of subnets. In a virtualized environment, this typically means that the appropriate networks are available to the virtual switches used for provisioning. Depending on the size of your environment and complexity of your network layout, you may need to do additional work on the virtual switch side and VM controller configurations to ensure that new servers are brought up on servers with access to the appropriate subnets. At Advance, where nearly all subnets were available to all VMs for provisioning, this was vastly simplified; in most organizations however this is not the case.
- Network service support - either pre-provisioned static IP addresses for new servers with appropriate ports provisioned, or DHCP. Since most bare-metal configuration requires DHCP and PXE booting capability, having both will make your life much easier. If a subnet fills up, your auto-deployment tools should be robust enough to capture and handle that error, even if only by paging an admin to resolve the problem. One of the reasons the Infoblox was terrific for this deployment was the ease of access to its DHCP interface for both querying of available addresses and provisioning of reserved addresses.
- DNS readiness for automated deployment. This means that your DNS zones should be laid out clearly, with reasonable reverse-mapping of IP addresses, so that automated provisioning is straightforward. The system needs to know what IP address to assign based on system role.
- Appropriate connectivity to build environments. You must have the bandwidth to push down OS images and patch data to multiple servers quickly.

- API or command-line access to your virtualization platform which will enable you to create new VMs, grab their MAC addresses, and hand information about them to your bare-metal deployment system. VMware is shaky in this regard, but it provided enough access for us to comfortably do what we needed.
- Automated OS licensing. If you need to enter a username and password at the console and that information can't be stored in an answer file, elastic expansion is a no-go.
- Automated patch management. This is often overlooked, but it's very important that a server brought up today look like one that was brought up last week. If we install an OS, even from the same image, but then run an update against current package repositories, our server today may have a very different set of packages from the server deployed last week. So it is important that all servers talk to the same repository set, with the same package version information across the board. We were struggling with this when I departed Advance, but had identified the Pulp project as a possible solution.

OS and Application Deployment

Your OS deployment choices will be largely shaped by your OS choice. As a CentOS environment, we used Cobbler for system deployments. There are a multitude of alternatives - Foreman, Spacewalk, or even hosting kickstart files on a regular webserver. The important thing is that the deployment system be able to identify a host and hand it the appropriate base configuration. Your OS install should be generic and minimal; don't try to handle 50 gold master images, but rather let your configuration management tool handle the heavy lifting.

At Advance, we chose Puppet as a configuration management system, and as I have since left Advance to work for Puppet Labs, my preferences are clear. However using any tool in this space puts your organization light years ahead of most of its competition. The key is not which configuration management tool you use, but the discipline to stick with that tool and keep everything in configuration management. Remember that, as discussed earlier, if it's not in configuration management, it can't be deployed automatically.

At this point I will return briefly to the concept of clusters that are not a collection of naive servers, but which must be aware of their own configuration or of each other. Configuration management provides the solution for this. Servers can be assigned environments or variables based on their intended role or position in a cluster, and configuration files can be templatized based on that information. In Puppet, we can use Exported Resources to ship dynamic information out of nodes to a shared datastore, so that other nodes can learn about them and make decisions. With proper scripting and policies, we can repartition our data sets in what is now a self-aware, elastically growing cluster.

Ad Hoc Administration

There are circumstances in any IT environment that don't fit well into the paradigm of change/configuration management. Suppose we want to kick all the Apache servers in a particular datacenter, or remount NFS volumes attached to a storage device that went belly-up?

The old solutions were SSH in a for loop, and ClusterSSH, which displays multiple terminals and allows a user to control them all simultaneously. Newer tools in this space

provide more accountability and control and better reporting.

At Advance we were using the Marionette Collective, or MCollective, for a few months when Puppet Labs acquired it, cementing our choice. Whether you using MCollective, func, fabric, Knife, or any other tool the important thing is that ad hoc administration should be compatible with your change management environment. If changes in one disrupt the other, automation will break. Many of these ad hoc tools force you into writing clients or carefully-wrapped agent scripts, something seen as an inconvenience. But there's a reason for this: we want to be able to execute something in a controlled period of time and then aggregate and return the results in a meaningful way. We can then store and report on the results and even audit the activities of the people using the tools.

The more centralized and automated this solution, the less likely it is to have unexpected impact on the managed environment. If we take the SSH in a for loop example - if we run that loop against 1500 servers, who is going to parse the results to notice that server 650's response didn't quite look right? And if it didn't, will the next round of changes cause server 650 to diverge even further from the remaining 1499? Tools with built-in auditing and data summarization can find these issues before they become problems or unexplained application behavior.

Where To Next?

I was saddened to leave Advance before we actually went elastic in production, but we had all the groundwork in place, thanks to the work of our infrastructure team's construction of their vDeploy tool, which interfaced with our VMware, DNS and DHCP environments to deploy new servers, then handed off to my Operations team's Cobbler and Puppet environments.

The workflow was that our Nagios-based monitoring system would trigger vDeploy only if the appropriate business criteria were met, causing vDeploy to build a new host based on information passed from Nagios. The concept of doing this sounded unthinkable at the start of the design process, but after analyzing the problem, it became clear that technologically, there were very few hurdles. Most applications and environments have APIs or RESTful interfaces that can be used for this sort of communication, and writing these scripts was simply a matter of putting in the work.

The actual complexity lay in building the application and business rules around when these things should happen. Focusing on communication and shared information rather than the engineering details proved to be the key. Good engineering and technology selection is key but is made much easier by taking the time to understand the business logic that these engineering exercises are designed to satisfy. While the impulse of many engineers is to jump in and start coding, taking the time to understand and manage the underlying cultural and infrastructure issues can turn development of an elastic environment from a seemingly insurmountable series of roadblocks to an exercise in small-scale script development.

Scaling on EC2 in a fast-paced environment

Practice and Experience Report

LISA 11

Nicolas Brousse, Lead Operations Engineer, TubeMogul, Inc.
Email: nicolas@TubeMogul.com

Abstract — **Managing a server infrastructure in a fast-paced environment like a start-up is challenging. You have little time for provisioning, testing and planning but still you need to prepare for scaling when your product reaches the tipping point. Amazon EC2 is one of the cloud providers that we experimented with while growing our infrastructure from 20 servers to 500 servers. In this paper we will go over the pros and cons of managing EC2 instances with a mix of Bind, LDAP, SimpleDB and Python scripts; how we kept a smooth working process by using NFS, auto-mount and shell-scripting; why we switched from managing our instances based on tailor-made AMI/Shell-scripting to the official Ubuntu AMI, Cloud-init and puppet; and finally, we will go over some rules we had to follow carefully to be able to handle billions of daily non-static http request across multiple Amazon EC2 regions.**

Index Terms - Amazon EC2, scalability, fault-tTubeMogulolerance, infrastructure, DevOps.

I. What is Amazon EC2 and how does it work?

Amazon AWS[1] provide a wide range of web-services. Amazon EC2[2] is part of AWS as a public cloud solution. EC2 let you start servers, called instances[3], on-demand. You are billed per-hour of usage and can stop an instance at any time. You can start your instance in a given geographic Region and Availability Zone[4].

Because of the large adoption of EC2, Amazon added a layer of indirection so that each AWS account's Availability Zones can map to different physical data center equivalents[5].

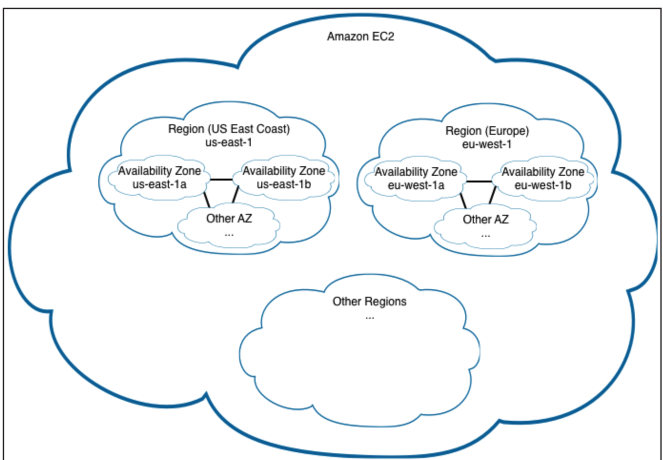


Fig 1. Amazon EC2 : Region and Availability Zone

When starting an instance, you will generally have to provide at least four pieces of information: the AMI[6] (server image), the instance type[7] (ram/CPU/arch), the Security Group[8] (firewall rules) and the Availability Zone. You can start an instance by using the Amazon EC2 API or the web console. By default an Amazon instance is started with some defined ephemeral storage space. Any data on it will be lost if you stop the instance. To use permanent storage you need to use solution like EBS. When stopping a server you lose the attached public and private IP. A new instance will have different IPs. The only way to keep a public static IP is to use Amazon EIP[9].

In September 2010, Amazon introduced some important features: Tagging, Filtering, Import Key Pair, and Idempotency. By adding customized tags (like hostname or profile name) you can easily filter your instances or EBS[10] volumes based on the given tags. In short, tagging and filtering lets you manage your own meta-information for each Amazon cloud resources.

II. Keep some order in your cloud

There are many client bindings built for the Amazon EC2 API which make it quite easy to use and implement. We started to use EC2 in 2008 by taking advantage of the computing ability that Amazon provide. We start a few dozen of servers for a few hours a day to fetch and aggregate data from different partners. The aggregated data are pushed into our shared MySQL cluster at our Colo center.

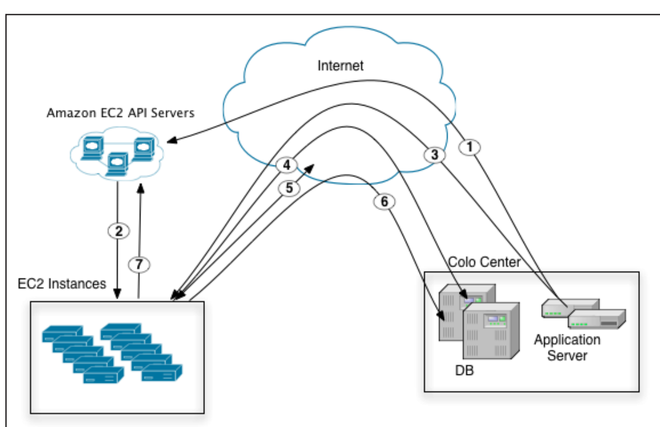


Fig 2. EC2 and Colo center

In Figure 2, you can see how we interact with EC2 to crawl our partners API and store data in our database. 1) our

application server calls the Amazon API at defined interval to start Amazon instances. 2) Amazon launch the instances we requested. 3) we push our code to the EC2 instances and start our program. 4) our application open an SSH tunnel to our databases. 5) we crawl our partner's API and aggregate the data as we want. 6) we write the results to our databases. 7) EC2 instances kill them-selves when they are done crawling.

This design works great and requires really low maintenance. Though, when you work in a startup environment, product evolve quickly. We needed to quickly develop our new video analytic product with a large number of servers to handle the analytics for billions of video stream per month. We chose to build this new product entirely on EC2. This let us to change the application quickly while the product grew without worrying about adding servers, rack, wiring, etc. Because of the nature of our product, we needed permanent storage, that's why we started to use EBS volumes.

To be able to add or remove nodes easily with different instance profiles it's important to be able to quickly identify what a server is doing and identify what its role is (Web server, Database, Hadoop namenode/datanode, etc.). To keep some order in our cloud we used clear security group, human readable hostnames (no ip-XXX.compute.internal or domU-XXX.compute.internal), NFS home directories and a strong and flexible monitoring.

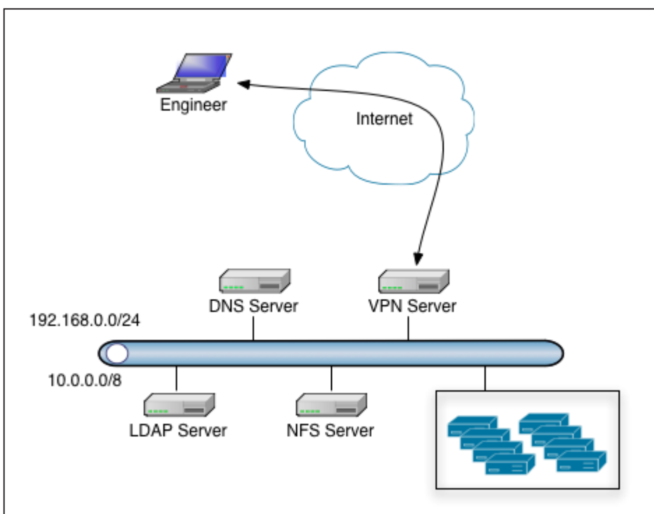


Fig 3. EC2 and our private network

A. Controlling access to the servers

1) Amazon EC2 Security Groups can get a bit cumbersome to manage especially when you want to access servers from anywhere without updating your rules while keeping a strong security policy. It's easy to forget to update or remove an old ip, etc. This is why we chose to manage our servers by setting up OpenVPN[11] servers on two of our Amazon instance using static IP, aka EIP. The ingress rules for our Security Groups stay simple by allowing SSH only from those VPN servers and by opening only the required public port if any. The VPN (using OpenVPN with auth-ldap[12] plugin) add another layer of security ensuring that only people with a valid username and password and a valid unique certificate can get access.

2) In addition to firewalls, we needed to give restricted access to some DBA, developers or contractor. Some needed root access. Our rule of thumb: *"You only get the permission you really need"*. No need to give root access to every server to your boss if he don't even know what to do with it. To manage those permissions and user accounts we used OpenLDAP[13]. All our instances are configured with pam_ldap. We extensively use pam_filters to grant access based on hostname, host group and Availability Zone.

```
pam_filter |(host=dev-mysql01.us-east-1b)(host=dev-mysql01.us-east-1)(host=dev-mysql01.\*)(host=dev-mysql\*.us-east-1b)(host=dev-mysql\*.us-east-1)(host=dev-mysql\*.\*)(host=\*.us-east-1b)(host=\*.us-east-1)(host=\*)
```

At any time we can grant or revoke access to any users for a server or multiple servers in one or multiple regions.

B. Identify running instances

Having obscure hostnames doesn't make your life easy when you start to deal with multiple instance profiles and multiple products with an extra-small sysop team (one or two people). When a product is in its early days with frequent changes, developers often needed access to the servers to be able to troubleshoot issues and find out why their last release wasn't working as expected. To help identify our hosts we used one of our EC2 instances as a management server configured with a DNS service (Bind[14]) patched for the ldap backend[15] and a LDAP service (OpenLDAP 2.4) using some of our own LDAP schema. For each host we stored in LDAP the private IP (10.0.0.0/8) and the public IP (it can be an EIP). Each host that we started used an AMI configured with the given private IP of the name server. Our resolv.conf would look like this:

```
domain <product>.private
search <product>.private <product>.public
nameserver 10.X.X.X
```

When starting an instance we also used the user-data to update the /etc/hostname. The user-data is an optional parameter you can use when starting an EC2 instance. This can support up to 16KB data. On the server you can fetch those user data at boot through an init script doing a curl command:

curl -s http://169.254.169.254/latest/user-data

From there, a lot become possible. In our case, we initially used the user-data just to pass our server hostname, example: "hostname=dev-mysql01". Note that, in the same way you can have access to many meta-data of your running instance:

curl -s http://169.254.169.254/latest/meta-data/

The pam ldap was configured to use the DNS entry to get the LDAP server IP.

uri ldaps://ldap.<product>.private

We started instances using a Java command line tool, called ec2ldap. We wrote it using Typica[16] (Java Binding for Amazon API), SQLite[17] and LDAP. We kept tracking of all our instances name and profiles in a SQLite database and used a script called Cerveza wrote in Tcl/Tk to access our hosts easily and do large maintenance with some one-liners:

./cerveza remote mysql[1-40] service mysql restart

With the SQLite database and Cerveza, it was easy for us to run over all our EC2 instances and update the resolv.conf if our management box went down and got a new IP. This worked well for a while but there were some important single point of failures[18] (SPOF) that finally bit us.

C. The benefit of NFS auto-mounted home directory

As stated earlier, developers needed easy access to the servers. To make their life easier we did setup an NFS export on our management box and used Autofs to mount the home directories on all our EC2 instances.

```
/etc/auto.master:
        /home   /etc/auto.home  intr,soft

/etc/auto.home:
        * fstype=nolock,noatime,soft,intr nfs.<zone>.private:/
home/&
```

This setup makes it easy to run a script across multiple instances without copying the instance to each host. It has been a great help in our dev environment but also when troubleshooting many servers in production. It's convenient, because you get your bash aliases or user script everywhere you login, etc. Unfortunately there is a downside, your access files can get slow, home dir can get stuck or permanently mounted if a service write to the home directory or keep a file descriptor open, etc.

In many cases we ended up using those auto-mounted home directories to run shared scripts on the first boot of an instance to deploy code, build our Raid devices with multiple EBS or reassemble them using mdadm or LVM.

D. Instance monitoring with Ganglia[19] and Nagios[20]

We choose to monitor our infrastructure with Nagios and Ganglia. It was a no-brainer for Nagios as we already used it to monitor our Colo servers and were quite used to its configuration. Ganglia was new for us as we used to graph our servers with Munin[21]. In our case, the decision between Munin and Ganglia was made on poll versus push model. Munin server poll each client, this requiring many resources on the main server especially when building each graphs. Ganglia uses a push model, each client report to the main process (gmond). Ganglia allow much more flexibility in graphing grids and clusters although we couldn't use the multicast support. For security purposes, Amazon EC2 doesn't let you to do multicast (or broadcast) on their network.

We configured multiple gmond processes on our management box to listen on different ports and collect data in different cluster group (one per Amazon Security Group) then just one gmetad process to collect all the data from each local gmond. This helped us to organize our graphs. Our EC2 instance were getting configured at first boot by running a ganglia configuration script that ensures the instance reports to the correct gmond process (if instance in SG dev, reports to port 8630, if SG mysql, report to 8631, etc.). Ganglia is a powerful solution so we were able to use the Python module to graph[22] our Java process using JMX[23] with JPype[24]. All those data are grouped in different dashboard and give us a quick way to spot issues.

For our Monitoring we use Nagios 3.2 with NSCA[25] and regex (in nagios.cfg: use_regexp_matching=1). We defined some generic service definitions for each cluster of servers. Some of our checks were directly looking at our RRD[26] data generated by Ganglia. Because of the quickly growing numbers of servers and services monitored we started to have too much I/O (read/write RRD files). We started to use rrdcached[27] which solved most of the problem but we still had many Nagios active checks which occasionally lead to swapping or slowness during checks. To fix the problem we simply split our ganglia load between two different management boxes, both servers use rrdcached to reduce IOs.

III. Learning the hard way

(or how to lock yourself out of your servers...)

While we were building our infrastructure and upgrading our network configuration, we were aware of few SPOF being introduced but they had a low impact or no impact on our production environment. However, what was initially designed for convenience and laziness became critical. The way we started to depend on those services make them even more critical. We didn't see it coming initially. This is the story of a three days nightmare starting with a VPN outage, then NFS/LDAP outage locking us out of all our EC2 instances.

A. The outage

1) For some reason, our file system storing our Nagios and Ganglia files were corrupted (EBS or Raid problem). This lead to many process getting stuck trying to access the faulty device. Too many resources were being used so the OOM Killer started killing processes, including our VPN process. After many reboots of the management server, nothing came back up. The console output showed a prompt for fsck check due to the faulty device. We had to kill the instance and start a new one.

2) The new instance failed to start. It prompted us again for fsck on our EBS volumes (used for NFS home dir). In fact, the mount point was defined in the fstab in the AMI, so it kept trying to mount the failing EBS with no way for us to fix it. There is no KVM with EC2, so we didn't have any way to try to recover from this situation. We ended up starting a new instance with an old AMI from which we removed the fstab so we could start the instance and finish it manually by running fsck, etc.

3) After reboot, our instance got a new Private IP allocated. This meant a new IP for our DNS, LDAP Producer and NFS. After recovering our instance we reimported our last ldif backup to LDAP. As the DNS server IP was hardcoded in our instance, we had to "manually" login on each server using a local account with the ssh keypair then update the resolv.conf, dnsmasq.conf, dhclient.conf, restart autofs and dhclient.

4) Unfortunately, as we used an old AMI for our management box, we lost many configuration settings breaking our Nagios and Ganglia services but also our command line tool (Cerveza) used to query our SQLite DB and easily access any hosts. This slowed our ability to recover a basic setup to be able to see what was wrong and fix it.

5) The ssh backdoor didn't always worked. We had to restart many instances manually. At boot they couldn't load our boot scripts from NFS. We had to login and finish the boot process manually by fixing Autofs then run the boot scripts. We also had to reconfigure many ssh tunnels, fix mysql replication, and recover missing or outdated configuration files, etc.

6) Some of the servers were using private IP in the EC2 Security Group, rebooting those server make the outage more complex as we needed to review all our security rules.

Luckily, this outage didn't affect our production services but it did lock us out of our servers for a long time. Needles to say, we took some time to revisit what went wrong and how we can fix it.

B. What we quickly fixed

1) One of the biggest pains during this outage, was our pam ldap and ssh configuration. Long timeout was preventing us from login into many servers (the cumul of timeout were higher than our ssh LoginGraceTime timeout, set to 2 min.), so the first thing was to reduce the autofs and ldap timeout and change nsswitch to look at the local account before ldap so even if our dns and ldap goes down, we still have an ssh backdoor to login and do local fix or maintenance.

```
/etc/auto.master :
   /home          /etc/auto.home timeout=5,retry=0,rw,intr,soft

/etc/nsswitch.conf:
   passwd:     files ldap
   shadow:    files ldap
   group:       files ldap

/etc/ldap.conf:
   timelimit 15
   bind_timelimit 5
```

2) We fixed our resolv.conf to handle better failover using: options attempts:1 timeout:1
3) We set up a better service and dns caching on each host using nscd instead of dnsmasq. We enabled caching for group, passwd, hosts and services.
4) We configured a secondary VPN service on our second management server and configured the OpenVPN clients to use "remote-random" option.
5) We stopped saving our fstab in the AMI so we could boot our instance even when a fsck is required.
6) We stopped using private IPs in our EC2 security group
7) We use a Haproxy[28] loadbalancer for DNS and LDAP service via Public IP using EIP.
8) Better version control of our boot scripts and AMI. We now manage almost everything with our configuration management tool.

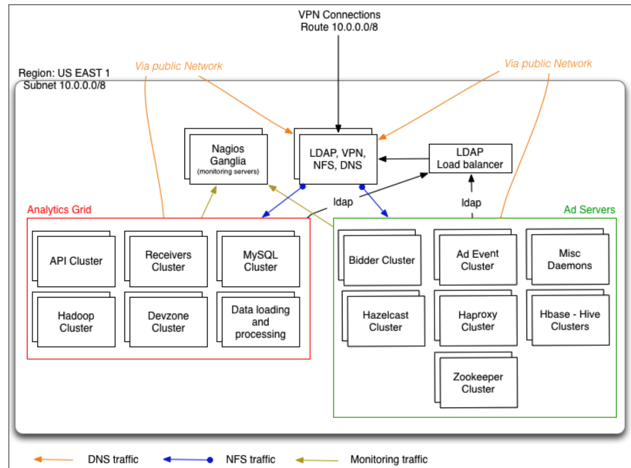


Fig 4. Network Flow between Clusters and Grid

IV. Going Worlwide

While our business evolved, we had a need to have a presence in different part of the world. This is easy to do with Amazon multiple region, though we have response time constraint with many partners. Our ninety-ninth percentile response time must be under 120 ms, including network round trip. Our partners are within 60 ms of our Amazon servers so it doesn't leave us much room especially if you consider the network variation inside Amazon's network or a noisy neighbor.

While building our international clusters, we tried to keep two goals in mind. First, how to reuse our existing tools and automate as much as we can. Second, do not create new SPOF failures in one region that would impact the others.

A. Simplify the instance boot process

With over 500 EC2 instances spread in multiple regions, we had to make our life easier. We got rid of our tool “ec2ldap” in Java and rewrote Cerveza in Python using Boto[29] (Amazon API binding for Python). We rewrote Cerveza to handle full instance start/stop/reboot with profile management. We chose Python over Java because of the scripting nature of Python. We didn’t want to slow ourselves down in a compile/release process for this simple tool. A scripting language lets us add features quickly and do quick bug fixing.

Our previous outage led us to stop using SQLite. We wanted a solution where we do not have to rely on a local database or to be forced to start/stop instances from a management server. We replaced SQLite for Amazon SimpleDB[30] to store only profile information. For the rest we leverage the Tagging feature of the Amazon API. All our hosts or EBS volumes are tagged with hostname, device name, etc. This gives us much more flexibility as we can run Cerveza from our own laptop. We are not depending on the location of our SQLite database, we can start, stop, reboot instances from anywhere for any kind of server we want to start. The other major thing we got rid is the home made AMI. It takes lot of time to build and maintain an AMI, so it’s not practical to deploy changes, etc. We chose to move to the official Ubuntu EC2 AMI and use cloud-init[31]. This is powerful. Cloud-init allow us to kick off our instance with different profiles by passing advanced user-data or scripts.

When starting a host with Cerveza for the first time we need to specify the instance profile we want to start (Hadoop node, MySQL, Java server, etc):

cerveza -m noc -- --zone ap-southeast-1a --start demo01 --profile UbuntuGeneric32Bit

To stop the host:

cerveza -m noc -- --zone ap-southeast-1a --stop demo01

To start the host a second time, we don’t need to define the profile again, cerveza know it by querying SimpleDB :

cerveza -m noc -- --zone ap-southeast-1a --start demo01

Besides using LDAP for DNS data and SimpleDB for profiles information of existing hosts, Cerveza also uses Yaml[32] to define our instances profiles and volume profiles.

```
--- !InstanceProfile
name: UbuntuGeneric32Bit
desc: Ubuntu Generic instance profile without EBS
Volumes
aws: !InstanceAws
   ami: { us-east-1: ami-a6f504cf, us-west-1:
ami-957e2ed0, ap-southeast-1: ami-7c423c2e, ap-
northeast-1: ami-3a0fa43b, eu-west-1: ami-339ca947 }
   security_group: devzone
   key_pair: tm-devzone
   type: c1.medium
   elastic_ip: false
volumes: [ ]
startup_scripts: [ ]
shutdown_scripts: [ shutdown ]
user_data: [ cloud-config-base.txt, setup-hostname.sh,
root-login.sh, cloud-config-puppet.txt ]
check_ec2_kernel: 2.6.35-28-virtual
```

Our Ubuntu Generic 32 Bit instance is generally used for development purpose. In this profile we just define some basic information (instance type, key pair, default SG, AMI, etc.) but also important user-data. By passing a list of files, Cerveza will automatically concat all the given file to generate a compressed mime-multipart data file and pass it in the user-data when launching the instance. Cloud-init will read it and execute each script when the server boot. Cloud-init allow advanced configuration and many possibilities. In our case, the user-data script cloud-config-puppet.txt let us configure Puppet[33], our configuration management tool, at boot time.

B. Use a configuration management tool

We were thinking about using a configuration management tool for a long time, but hesitated until LISA 10. As we changed our AMI and started to use cloud-init, we took the opportunity to deploy puppet on all our hosts and start using it. We briefly looked at Cfengine[34] and Chef[35] too, but finally decided to go with Puppet as it seemed a little more documented and already fully integrated to Cloud-init.

Configuring and deploying puppet is fast and easy but using it properly is not that obvious. We had to deal with a couple of annoying problems like huge CPU spikes on each client, obscure errors for non-initiate people, process not running because of a lock file after reboot, etc. We addressed most of those issues. We found out that abusing of Augeas[36] is not necessarily good. We were able to speed up our puppet run from over 400 seconds to less than 15 seconds by replacing Augeas by puppet templates (mostly on long sysctl configuration). We use some ruby environment variables[37] to optimize each puppet client run, though we are still experimenting those. We stopped running puppet as a daemon as “fileserver” used too much resources. We had cases were puppet was using over 1GB of ram leading OOM Killer to kill some other process like our Membase[38] server. We now setup our puppet in a crontab running every half an

hour. To avoid a peak of requests on our puppet master we run the cron at random minutes on each client.

```
# schedule puppet to run via cron
$minute1 = generate('/usr/bin/env', 'sh', '-c',  'printf $((RANDOM
%29+0))')
cron {
   "puppet_run":
      ensure      => present,
      command     => "/usr/sbin/puppetd --onetime --no-daemonize --
logdest syslog > /dev/null 2>&1",
      environment => [ 'RUBY_HEAP_MIN_SLOTS=500000',
                       'RUBY_HEAP_SLOTS_INCREMENT=250000',
                       'RUBY_HEAP_SLOTS_GROWTH_FACTOR=1',
                       'RUBY_GC_MALLOC_LIMIT=500000'
                     ],
      user        => "root",
      minute      => $minute1,
      hour        => "*";
}
```

In the end, Puppet makes our life easier to manage and change configuration on multiple servers in four different data centers. Our puppet masters are located in our Colo center on US east coast. They are setup with Apache 2 + Phusion Passenger[39] with one master and one failover server. The failover server also handles the puppet reports using Puppet Dashboard[40]. We patched the puppet clients to report their FQDN as hostname instead of using there certificate name.

We currently don't have a clear dev environment for our puppet configuration, though our dev servers are setup to use a different environment so we can test our modules changes in dev before pushing to production. We are looking at better ways to manage this.

in puppet.pp:

```
class puppet inherits puppet::init {
   if $hostname =~ /^dev-*$/ or $ec2_security_groups ==
"devzone" {
      augeas {
         "puppet_env":
            context => "/files/etc/puppet/puppet.conf/main",
            onlyif  => "get environment != 'development'",
            changes => "set environment 'development'",
            notify  => Exec["puppet"];
      }
   }
}
```

in puppet.conf:

```
[development]
   manifestdir    =   $confdir/dev/manifests
   manifest       =   $manifestdir/site.pp
   modulepath     =   $confdir/dev/modules:$confdir/modules
```

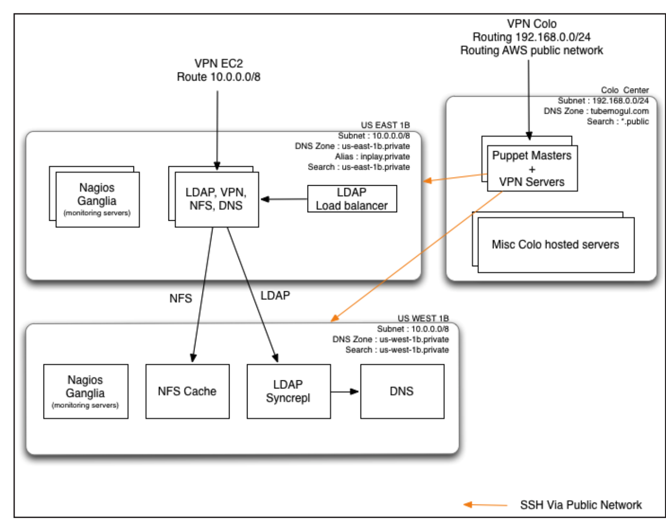


Fig 5. Network Flow between multiple AWS regions

C. Mirroring DNS, LDAP, NFS

Because of the multi-region and our response time constraint, we had to get DNS servers on each region. We use some "gateway" servers whose role is to serve as local DNS server, LDAP and NFS. As our DNS depend on LDAP, we initially setup LDAP Proxy with query caching which was working great except when running a non-cached query. We were getting some latency spike of up to four seconds for a DNS response. This was affecting our production response time in some cases increasing our percentage of timed out requests. We changed this configuration to use LDAP syncrepl[41]. Each LDAP server on each region is a master replicating one of our master server on US EAST. This solved our DNS response time and pam ldap response time. Though, since we use Autofs for our home directories we had to address the problem for our NFS server. On each region we use a NFSv4 mount with FS-Cache (cachefilesd[42]), this aimed to improve read speed on each region. The key thing we did was to remove the NFS mount point from the updatedb configuration because it would generally kill the server performance.

```
/etc/updatedb.conf:

PRUNE_BIND_MOUNTS="yes"
PRUNEPATHS="/tmp /var/spool /media /opt/openldap/var /
EBS /home"
PRUNEFS="NFS nfs nfs4 rpc_pipefs afs binfmt_misc proc
smbfs autofs iso9660 ncpfs coda devpts ftpfs devfs mfs shfs sysfs
cifs lustre_lite tmpfs usbfs udf fuse.glusterfs fuse.sshfs ecryptfs
fusesmb devtmpfs bindfs"
```

We are still not fully satisfied of our current solution and may stop using NFS for our home directory as it introduces a possible snowball effect in case our NFS fails on US east. Auto-mounted home directory doesn't give us any more added value as the product matures and our server

infrastructure grows. Also, we are having more clients using the NFS doing multiple mount/unmount leading to frequent home directories being stuck with a "Stale NFS file handle"[43].

D. What else?

To speed up our application deployment in multiple regions we started to use Amazon S3[44] with localized buckets. Instead of pushing our files from our Colo to each server, we push the files once to each of the localized S3 buckets then fetch the files to release on S3 from each server and deploy them locally.

Overall, with this infrastructure, we still have room for many improvement:

1) One clear blocker is NFS, we definitely plan to entirely remove NFS with auto-mounted home directory and get back to a more standard way to manage our servers. We are introducing more security checks and rules limiting production access so there shouldn't be any more need of user home directory being synchronized this way on all our servers.
2) We currently have two different sets of VPN and LDAP servers, one in our Colo and one in EC2. We want to centralize them to simplify our user and ACLs management.
3) We still have some "Gateway" servers, doing bridge between our regions. They are not based on the Ubuntu EC2 AMI. For lower maintenance on our side, we want to migrate everything onto the official Ubuntu EC2 AMI and fully use Cloud-init possibilities. We also want to get to a more standardized approach of managing our setup by using our internal Debian repository when required.
4) We are looking at Amazon VPC[45] to be able to better manage our private IPs and clusters. It can help to have better security policies in place preventing your backend from being accessed int the public internet, etc.
5) We plan to look again at Amazon ELB[46] to manage our different load balancing. One of the biggest drawbacks we had with ELB was the lake of visibility. No access logs and no clear error reporting make things hard to troubleshoot especially when you start having 500 errors returned by ELB during traffic spike.

V. Lesson learned

Evolution of your infrastructure must stay fault-tolerant in any case. What was simple and working at first can get complex in a multi-region / high latency environments.

In a small team with limited resources you will have little time to get everything right. You will miss important point leading to outages. Make sure to have a valid backup strategy and have a recovery procedure.

Never build a SPOF, even if it's for a "non-critical" use. As you start to rely more on this services (and you generally don't see it coming), your SPOF can have more impact than you would anticipate.

Infrastructure legacy can become a pain to maintain. Don't be afraid to revisit what you did and change it. What was true at one point of your design may not be true anymore.

Scaling your infrastructure in a fast paced environment require a lot of automation. which is why using a configuration management tool early would prevent you many headaches later on.

Acknowledgments

I would like to thank the LISA Chair and my shepherd, Marc Staveley, for the opportunity of this paper. It's an insightful experience that I would not hesitate to recommend to anyone.

I also want to thanks my close friends and family who continuously support me in my career choices.

This paper wouldn't have been possible without the opportunity I got by moving to the USA and joining TubeMogul in 2008 after just few Skype interviews. Hence, I express all my respect and consideration to John Hughes and Brett Wilson, TubeMogul's Founders.

References

[1] **Amazon Web Service (AWS)**
Amazon Web Services (AWS) delivers a set of services that together form a reliable, scalable, and inexpensive computing platform "in the cloud".
Website: http://aws.amazon.com

[2] **Amazon Elastic Cloud (EC2)**
Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.
Website: http://aws.amazon.com/ec2

[3] **Amazon Instance**
An Amazon Instance is the AWS version of a server. It's known to be a Xen DomU Virtual Machine. Instances come in a variety of configurations and are designed to provide predictable and dedicated computing power on demand.

[4] **Availability Zone (AZ) and Regions**
Amazon EC2 provides the ability to place instances in multiple locations. Amazon EC2 locations are composed of Availability Zones and Regions. Regions are dispersed and located in separate geographic areas (US, EU, etc.). Availability Zones are distinct locations within a Region that are engineered to be isolated from failures in other Availability Zones and provide inexpensive, low latency network connectivity to other Availability Zones in the same Region.

[5] **Matching EC2 Availability Zone Across AWS Account**
By Eric Hammond on July 28, 2009
"*Summary*: EC2 availability zone names in different accounts do not match to the same underlying physical infrastructure. This article explains a trick which can be used to figure out how to match availability zone names between different accounts."
Blog post: http://alestic.com/2009/07/ec2-availability-zones

[6] **Amazon Machine Image (AMI)**
An Amazon Machine Image (AMI) is an encrypted machine image stored in Amazon S3. It contains all the information necessary to boot instances of your software.

[7] **Amazon Instance Type**
A specification that defines the memory, CPU, storage capacity, and hourly cost for an instance. Some instance types are designed for standard applications while others are designed for CPU-intensive applications.
Link: http://aws.amazon.com/ec2/instance-types

[8] **Amazon Security Group (SG)**
A security group is a named collection of access rules. These access rules specify which ingress (i.e., incoming) network traffic should be delivered to your instance. All other ingress traffic will be discarded.

[9] **Amazon Elastic IP (EIP)**
Elastic IP addresses are static IP addresses designed for dynamic cloud computing. An Elastic IP address is associated with your AWS account not a particular instance, and you control that address until you choose to explicitly release it. Unlike traditional static IP addresses, however, Elastic IP addresses allow you to mask instance or Availability Zone failures by programmatically remapping your public IP addresses to any instance in your account.

[10] **Amazon Elastic Block Store (EBS)**
Amazon Elastic Block Store (EBS) provides block level storage volumes for use with Amazon EC2 instances. Amazon EBS volumes are off-instance storage that persists independently from the life of an instance. Amazon Elastic Block Store provides highly available, highly reliable storage volumes that can be attached to a running Amazon EC2 instance and exposed as a device within the instance.
Website: http://aws.amazon.com/ebs

[11] **OpenVPN** "is a free and open source software application that implements virtual private network (VPN) techniques for creating secure point-to-point or site-to-site connections in routed or bridged configurations and remote access facilities. It uses SSL/TLS security for encryption and is capable of traversing network address translators (NATs) and firewalls." in Wikipedia: The Free Encyclopedia.
Website: http://openvpn.net

[12] **Auth-LDAP** plugin for OpenVPN
Website: http://code.google.com/p/openvpn-auth-ldap

[13] **OpenLDAP** is an open source implementation of the Lightweight Directory Access Protocol.
Website: http://www.openldap.org

[14] **BIND** is by far the most widely used DNS software on the Internet. It provides a robust and stable platform on top of which organizations can build distributed computing systems with the knowledge that those systems are fully compliant with published DNS standards.
Website: http://www.isc.org/software/bind

[15] Our Bind 9 install is patched with bind9-ldap + internal patch to support our LDAP schemas and specifics EC2 needs.
Website: http://bind9-ldap.bayour.com

[16] **Typica** is Java client library for a variety of Amazon Web Services.
Website: http://code.google.com/p/typica

[17] **SQLite** is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.
Website: http://www.sqlite.org

[18] **Single Point Of Failure (SPOF)**
"A single point of failure (SPOF) is a part of a system that, if it fails, will stop the entire system from working.[1] They are undesirable in any system with a goal of high availability, be it a network, software application or other industrial system. Systems are made robust by adding redundancy in all potential SPOF and is generally achieved in computing through high-availability clusters. Redundancy can be achieved at the internal component level, at the system level (multiple machines), or site level (replication)." in Wikipedia: The Free Encyclopedia.

[19] **Ganglia** "is a scalable distributed system monitor tool for high-performance computing systems such as clusters and grids. It allows the user to remotely view live or historical statistics (such as CPU load averages or network utilization) for all machines that are being monitored." in Wikipedia: The Free Encyclopedia.
Website: http://ganglia.info

[20] **Nagios** "is a popular open source computer system and network monitoring software application. It watches hosts and services, alerting users when things go wrong and again when they get better." in Wikipedia: The Free Encyclopedia.
Website: http://www.nagios.org

[21] **Munin** is a networked resource monitoring tool that can help analyze resource trends and "what just happened to kill our performance?" problems. It is designed to be very plug and play. A default installation provides a lot of graphs with almost no work.
Website: http://munin-monitoring.org

[22] **Graphing Java JMX Object values with Ganglia and Python using JPype**
Blog post: http://goo.gl/LL7X3

[23] **Java Management Extensions (JMX)**
Set of specifications for application and network management in the J2EE development and application environment

[24] **JPype** is an effort to allow python programs full access to java class libraries. This is achieved not through re-implementing Python, as Jython/JPython has done, but rather through interfacing at the native level in both Virtual Machines.
Website: http://jpype.sourceforge.net

[25] **Nagios Service Check Acceptor (NSCA)**
NSCA allows you to integrate passive alerts and checks from remote machines and applications with Nagios. Useful for processing security alerts, as well as deploying redundant and distributed Nagios setups.
Website: http://goo.gl/ikagM

[26] **RRDtool** is the OpenSource industry standard, high performance data logging and graphing system for time series data. RRDtool can be easily integrated in shell scripts, perl, python, ruby, lua or tcl applications.
Website: http://oss.oetiker.ch/rrdtool

[27] **rrdcached** is a daemon that receives updates to existing RRD files, accumulates them and, if enough have been received or a defined time has passed, writes the updates to the RRD file.
Website: http://oss.oetiker.ch/rrdtool/doc/rrdcached.en.html

[28] **Haproxy** is a "Reliable, High Performance TCP/HTTP Load Balancer"
Website: http://haproxy.1wt.eu

[29] **Boto** is a Python interface to Amazon Web Services
Website: http://code.google.com/p/boto

[30] **Amazon SimpleDB (SDB)**
Amazon SimpleDB is a highly available, flexible, and scalable non-relational data store that offloads the work of database administration. Developers simply store and query data items via web services requests, and Amazon SimpleDB does the rest.
Website: http://aws.amazon.com/simpledb

[31] **Cloud-init** is the Ubuntu package that handles early initialization of a cloud instance. It is installed in the UEC Images and also in the official Ubuntu images available on EC2.
Website: https://help.ubuntu.com/community/CloudInit

[32] **YAML** is a human friendly data serialization standard for all programming languages.
Website: http://yaml.org

[33] **Puppet** is an open source configuration management tool.
Website: http://puppetlabs.com

[34] **CFEngine** automates IT processes and ensures the availability and consistency of applications and services.
Website: http://cfengine.com

[35] **Chef** is an open-source systems integration framework built specifically for automating the cloud. No matter how complex the realities of your business, Chef makes it easy to deploy servers and scale applications throughout your entire infrastructure. Because it combines the fundamental elements of configuration management and service oriented architectures with the full power of Ruby, Chef makes it easy to create an elegant, fully automated infrastructure.
Website: http://www.opscode.com/chef

[36] **Augeas** is a configuration editing tool. It parses configuration files in their native formats and transforms them into a tree. Configuration changes are made by manipulating this tree and saving it back into native config files.
Website: http://augeas.net

[37] **Fine tuning your garbage collector** By Chris Heald on June 13, 2009
Blog post: http://goo.gl/5GYBL

[38] **Membase** Server is the lowest latency, highest throughput NoSQL database technology on the market. When your application needs data, right now, it will get it, right now. A distributed key-value data store, Membase Server is designed and optimized for the data management needs of interactive web applications, so it allows the data layer to scale out just like the web application logic tier – simply by adding more commodity servers.
Website: http://www.couchbase.org/membase

[39] **Phusion Passenger**, aka mod_rails or mod_rack, allow easy and robust deployment of Ruby on Rails application on Apache and Nginx Webservers.
Website: http://www.modrails.com

[40] **Puppet Dashboard** is a web interface and reporting tool for your Puppet installation. Dashboard facilitates management and configuration tasks, provides a quick visual snapshot of important system information, and delivers valuable reports. In the future, it will also serve to integrate with other IT tools commonly used alongside Puppet.
Website: http://puppetlabs.com/puppet/related-projects/dashboard

[41] The **LDAP Sync Replication engine**, syncrepl for short, is a consumer-side replication engine that enables the consumer LDAP server to maintain a shadow copy of a DIT fragment. A syncrepl engine resides at the consumer and executes as one of the slapd(8) threads. It creates and maintains a consumer replica by connecting to the replication provider to perform the initial DIT content load followed either by periodic content polling or by timely updates upon content changes.
Documentation: http://www.openldap.org/doc/admin24/replication.html

[42] The **cachefilesd** daemon manages the cache data store that is used by network filesystems such a AFS and NFS to cache data locally on disk.
Man page: http://linux.die.net/man/8/cachefilesd

[43] **Stale NFS file handle**
Note: http://sysunconfig.net/unixtips/stale_nfs.txt

[44] **Amazon Simple Storage Service (S3)**
Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.
Website: http://aws.amazon.com/s3

[45] **Amazon Virtual Private Cloud (VPC)**
Amazon Virtual Private Cloud (Amazon VPC) lets you provision a private, isolated section of the Amazon Web Services (AWS) Cloud where you can launch AWS resources in a virtual network that you define. With Amazon VPC, you can define a virtual network topology that closely resembles a traditional network that you might operate in your own datacenter. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.
Website: http://aws.amazon.com/vpc

[46] **Amazon Elastic Load Balancing (ELB)**
Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances. It enables you to achieve even greater fault tolerance in your applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic. Elastic Load Balancing detects unhealthy instances within a pool and automatically reroutes traffic to healthy instances until the unhealthy instances have been restored. You can enable Elastic Load Balancing within a single Availability Zone or across multiple zones for even more consistent application performance.
Website: http://aws.amazon.com/elasticloadbalancing

DarkNOC: Dashboard for Honeypot Management

Bertrand Sobesto, Michel Cukier
Clark School of Engineering
University of Maryland
College Park, MD, USA
{bsobesto, mcukier}@umd.edu

Matti Hiltunen, Dave Kormann, Gregg Vesonder
AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
{hiltunen, davek, gtv}@research.att.com

Robin Berthier
Coordinated Science Laboratory
Information Trust Institute
University of Illinois
Urbana-Champaign, IL, USA
rgb@illinois.edu

Abstract

Protecting computer and information systems from security attacks is becoming an increasingly important task for system administrators. Honeypots are a technology often used to detect attacks and collect information about techniques and targets (e.g., services, ports, operating systems) of attacks. However, managing a large and complex network of honeypots becomes a challenge given the amount of data collected as well as the risk that the honeypots may become infected and start attacking other machines. In this paper, we present DarkNOC, a management and monitoring tool for complex honeynets consisting of different types of honeypots as well as other data collection devices. DarkNOC has been actively used to manage a honeynet consisting of multiple subnets and hundreds of IP addresses. This paper describes the architecture and a number of case studies demonstrating the use of DarkNOC.

1 Introduction

Because of the value of the data they store and the resources they provide, information systems become targets for attackers and must be protected. To better secure computer systems from external threats, security researchers aim to understand attackers and the different techniques they use to compromise computers and achieve their goals. One possible approach is to use a target computer, called a honeypot, which is not used by normal users. Therefore, all the activity towards this computer can be considered malicious.

Individual honeypots or networks of honeypots have been used to conduct various studies of attackers [1, 9] and analysis of cyber crimes such as unsollicited electronic mails, phishing [10], identity theft and denial of service. The computer security community has used honeypots to analyze different techniques deployed by the attackers to reach their objectives. Attackers' arsenal includes distributed denial of service [24], botnets [2], worms [11] or SPAM [15]. However few studies focus on the usage of honeypots data to help network administrators to better protect their production networks. Honeypot deployment is challenging and the architecture of such networks is complex. For example, distributed honeynets require secure tunnels and different levels of protection must be in place to ensure a total containment of attacks targeting the honeypots. In addition, honeynets require constant monitoring to guarantee that protection systems (for example firewalls, traffic shappers) and data collection are operating correctly. Depending on the size of the honeynet, the volume of data collected can be important and impacts significantly data processing and extraction. To be integrated as a security tool, honeypots data must be presented and translated in meaningful way to network administrators.

In this paper, we introduce DarkNOC, a solution designed to efficiently process large amount of malicious traffic received by a large honeynet, and to provide a user-friendly Web interface to highlight potential compromised hosts to security administrators, as well as to provide the overall network security status. DarkNOC is used to manage the UMD honeynet, a network of 2,000 honeypots from which information about attacks is continuously extracted and provided to the security team to help them better protect the production network.

The rest of the paper is organized as follows. In Section 2, we provide an overview of the architecture and operation of DarkNOC. In Section 3, we describe the outputs and views provided by the DarkNOC. We provide a number of case studies using DarkNOC in Section 4. Finally we review the related work in Section 5, we provide some remarks on future work in Section 6 and conclude the paper in Section 7.

2 DarkNOC Architecture

This section describes what DarkNOC does, how it collects data, and its internal structure.

2.1 System Architecture

DarkNOC manages multiple types of honeypots and information sources as illustrated in Figure 1. The UMD honeynet consists of low interaction honeypots (LIHs) such as Nepenthes [3] as well as high-interaction honeypots (HIHs) consisting of virtual or physical machines running real operating systems, applications, and services [5]. The UMD honeynet supports multiple subnets consisting of IP addresses contributed by different organizations participating in the research. DarkNOC collects multiple sources of information from different devices (e.g., NetFlow from Gateway, Snort events from Snort Sensors [20], and malware from Nepenthes), analyzes the data, and presents it to users in an efficient and actionable manner. The details of the data views and their use in analyzing security incidents are discussed in Sections 3 and 4.

The current information sources consist of the following:

- NetFlow Data: DarkNOC uses nfdump[1] to extract NetFlow data collected on the main gateway of the honeypots. The flow data provides enough information to determine the number of attackers, the different source and destination IP addresses, and the different source and destination ports. Specifically, each NetFlow record summarizes communication between two network end points (defined by the IP addresses and port numbers of the end points) including the time, duration, and numbers of bytes and packets (see example below), but does not contain any payload information (i.e., content of the messages transmitted).

```
Date flow start      Duration  Port  Src IP:Port  -> Dst IP:Port  Packets Bytes Flows
2010-02-09 06:43:... 4294966.937 TCP  218.8.251.187:20347 -> x.x.x.x:80  2  94   1
2010-02-09 06:43:... 4294966.977 TCP  218.8.251.187:20347 -> x.x.x.x:80  2  94   1
```

[1] http://nfdump.sourceforge.net/

- Snort Events: Snort [20] is an Intrusion Detection System (IDS) for detecting attacks and potential intrusions. Snort provides information about the types of attacks used against the honeypots.

- Malware Collection: Nepenthes acts as a passive malware collector by emulating common service vulnerabilities and allowing attackers to inject the malware binaries. Nepenthes provides a log of each malware submission containing information such as the date and the vulnerability used but also the binary injected. This allows DarkNOC to see what kinds of malware are successfully uploaded, the security signatures, and port used. It also allows to measure the efficiency of the security solution protecting the network.

2.2 DarkNOC Software Architecture

The design of the DarkNOC software architecture was driven by the following constraints:

- The aesthetics from the user's point of view: The user interface should be easy to access and the important data should be automatically highlighted. This interface should be highly portable so that users can use different operating systems and access the system from different geographic locations (i.e., not tied to one dedicated machine).

- Speed: The user interface must be fast and the user should not have to wait for the results to be displayed. Processing high volumes of data can be time consuming and if the processing is started only when the user requests a data view, the response time may not be satisfactory. Therefore, our system uses data pre-processing when possible to ensure fast response.

- Data validity: The data displayed should be reasonably up to date and reflect the current activity.

To meet these requirements, the application software has been divided into three different parts: 1) a graphical Web front-end, 2) back-end, and 3) alerting module. The front-end generates a Web page displaying the different information. The back-end extracts the necessary data from the flows and creates the different graphs.

Back-end Module: Written in Perl, the back-end module is a background process that updates the information displayed by the front-end every 5 minutes based on the NetFlow data. The separation of flow processing from the display was necessary to guarantee a fast response time at the user interface, because the extraction of flow

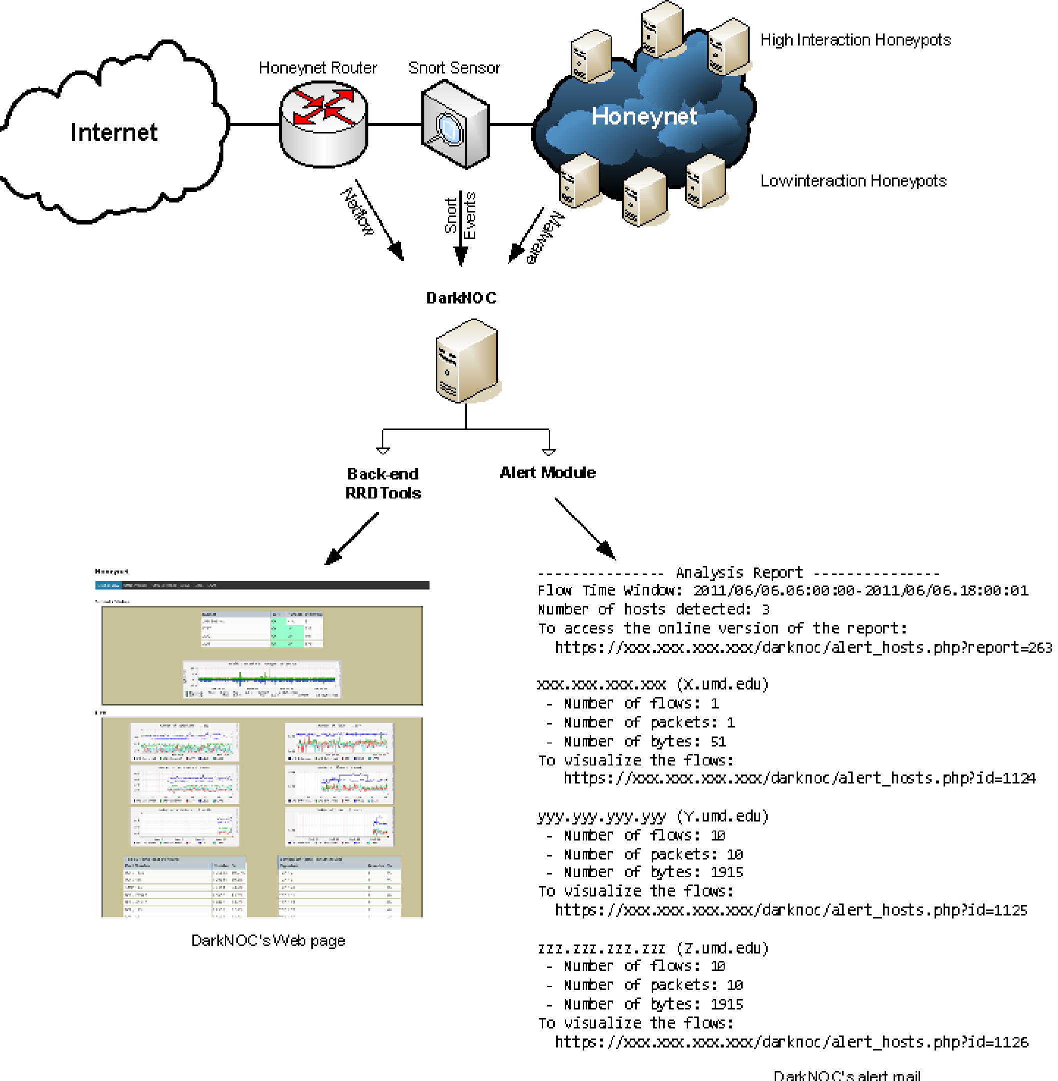


Figure 1: System architecture

data can be time consuming. Since the flow data is updated every 5 minutes by the flow collector, a continuous live update of the displayed views is unnecessary. However it requires the tool to process the new flow files within 5 minutes. DarkNOC provides information for the last 24 hours and the last 5 minutes. Two different processes generate the 24 hours and 5 minutes statistics. For about 2,000 IP addresses, an average of 15,995 flows are generated every 5 minutes representing about 5 million flows per day. It takes an average of 7.4 seconds to process a newly created flow file. Given this number, DarkNOC is able to process almost a hundred times more flows within 5 minutes. Generating the statistics on the last 24 hours is computationally more expensive and longer. It takes an average of 130 seconds. However, it is not necessary for this process to finish within 5 minutes. A lock file prevents multiple executions of this process at the same time. For each subnet and the global view, the back-end generates the different graphs, the list of destination ports, the list of attackers and the list of targeted honeypots. The graphs are created using RRDTool[2], an open source tool for storage and retrieval of time series.

Graphical User Interface: The graphical user interface organizes the different data necessary to present a summary of the honeypots activity. Web technologies such as the PHP language and Cascading Style Sheets are used. A Web page is extremely portable and requires no configuration on the client side. Figure 2 shows the homepage of DarkNOC. The content is described in Section 3. The graphical user interface first provides a global

[2] http://oss.oetiker.ch/rrdtool/

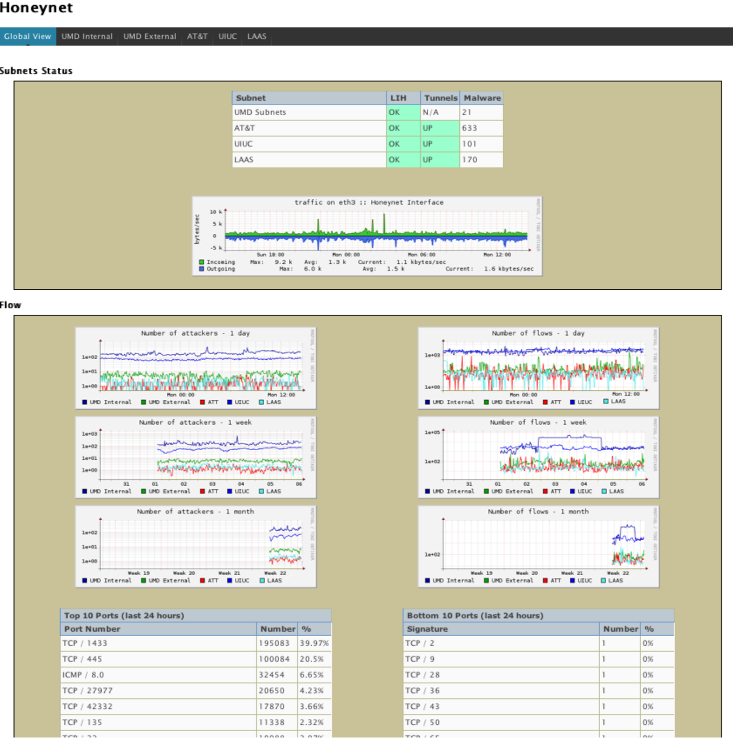


Figure 2: DarkNOC's graphic user interface

view of the activity of the honeypots: the data displayed includes all the subnets. The user has then the possibility to reduce the scope of analysis to one subnet. To prevent unauthorized access, the application uses an HTTP authentication over SSL to protect DarkNOC's directory on the Web Server. Apache is configured to authenticate users against an LDAP server where all accounts are centralized. User objects belonging to the group *DarkNOC* have access to the application. Because of legal and confidentiality reasons it is necessary to filter the information displayed by DarkNOC. Once authenticated DarkNOC retrieves the user name stored in the `$_SERVER['PHP_AUTH_USER']` variable and matches it with the user's table in the database to determine which subnets to display or not. If the user is allowed to access more than one subnet, DarkNOC will reflect the user's rights in the global view but also in the subnet selector. If the user has access to a single subnet, the subnet will be automatically selected with no possibility to select another one.

Alerting Module: The alerting module is a process executing a specific query on the flow data. The results are sent by email to a specific group of users. Users have the possibility to create their own flow query based on the nfdump filter syntax and to specify the recipients of the alerts. The module is currently launched twice a day: at 6:00 AM and at 6:00 PM. It can be executed more frequently if more real-time alerts are required.

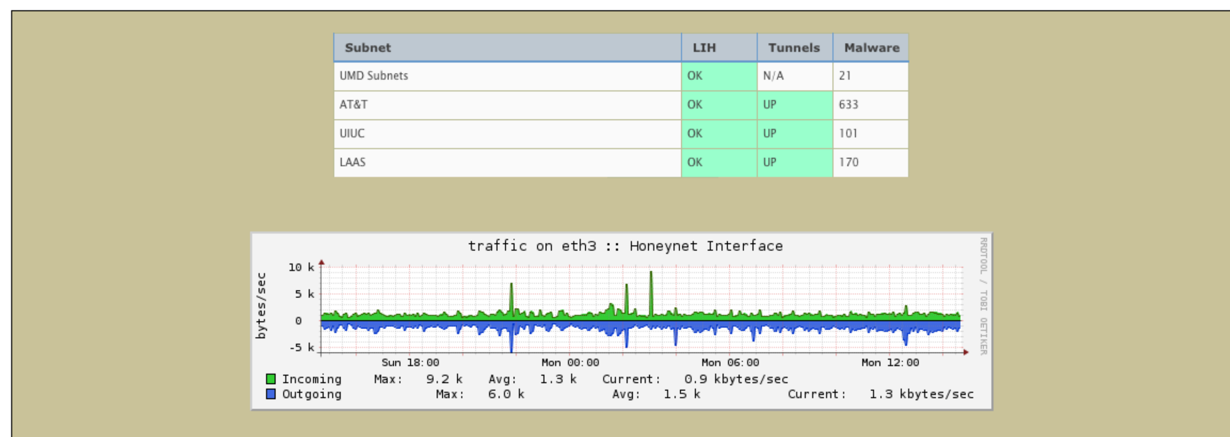
3 Display Description

The layout of the graphical user interface of DarkNOC presented in Figure 2 organizes the different pieces of information gathered from the most global and important to the most detailed concerning the current activity of the honeypots. The user interface of DarkNOC has been developed to ease the comparison of the different sources of information and the comparison of the different subnets.

The Web page provided by DarkNOC is divided into three different sections: 1) status of the subnets, 2) flow-based information, and 3) Snort events. Each section will provide information that will reduce the number of possible explanations when an anomaly in the traffic is identified in DarkNOC. The first screen provided is a global view of the honeypots activity. The user can select a specific subnet to drill-down to a more detailed view of the subnet activity.

3.1 Subnet Status and Network Traffic

The first part of the Web page shown in Figure 3 is composed of a table giving the status of the low interac-


Subnets Status

Subnet	LIH	Tunnels	Malware
UMD Subnets	OK	N/A	21
AT&T	OK	UP	633
UIUC	OK	UP	101
LAAS	OK	UP	170

Figure 3: Subnets status section

Top 10 Ports (last 24 hours)		
Port Number	Number	%
TCP / 1433	198939	40.03%
TCP / 445	101807	20.49%
ICMP / 8.0	32848	6.61%
TCP / 27977	21520	4.33%
TCP / 42332	18191	3.66%
TCP / 135	11563	2.33%
TCP / 22	10105	2.03%
TCP / 80	5204	1.05%
UDP / 5060	5119	1.03%
TCP / 139	4914	0.99%

Bottom 10 Ports (last 24 hours)		
Signature	Number	%
TCP / 2	1	0%
TCP / 9	1	0%
TCP / 28	1	0%
TCP / 36	1	0%
TCP / 43	1	0%
TCP / 50	1	0%
TCP / 65	1	0%
TCP / 66	1	0%
TCP / 67	1	0%
TCP / 71	1	0%

Figure 4: Top and bottom 10 transport ports targeted

tion honeypots (LIH) running Nepenthes, the status of the tunnels to different organizations, and the number of malware collected for each subnet since the initialization of DarkNOC. The notion of a tunnel is specific to the UMD honeynet. It allows to redirect the network traffic from remote locations to the honeypot network transparently. Hence, it is possible to use other participating organizations' IP addresses. A graph representing the incoming and outgoing traffic in bytes per seconds is included in the status section as well. This section provides essential indications on the state of the main components of the UMD honeynet, i.e. tunnels and main gateway. The graph gives an overview of the UMD honeynet infrastructure load and can help to detect anomalies in the traffic.

3.2 NetFlow Data

The NetFlow section provides information extracted from the NetFlow data collected at the edge of the honeypots network. Figure 5 presents a graph showing the number of attackers over time for each subnet of the honeypot network. Each unique IP address that does not belong to the honeypots is considered a unique attacker. The graphical user interface provides several graphs that display the number of attackers at different time scales: one day, one week, and one month. Figure 6 presents a graph showing the number of flows over time for each subnet of the honeypot network. Separate graphs are used to display the number of flows at different time scales.

These two graphs shown in Figures 5 and 6 make it easy to observe the activity of the honeypots for each subnet. Comparing the numbers of flows and attackers can reveal attack characteristics. For example, an increase of the number of flows while the number of attackers remains relatively steady means that one or several offenders may have launched an attack that generates large amounts of flows such as port scanning and brute-force activities. It can also mean that a large network behind a network address translation system is compromised and targeting the UMD honeynet. DarkNOC also makes it easy to compare trends between the different

subnets. For example, it is straightforward to identify peaks in the number of attackers or flows that occur at the same time in different subnets, as well as changes in the attacks directed to only one of the subnets, indicating a targeted attack.

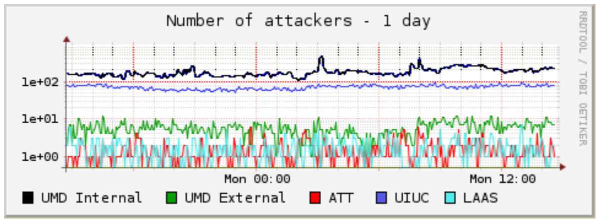


Figure 5: Number of attackers

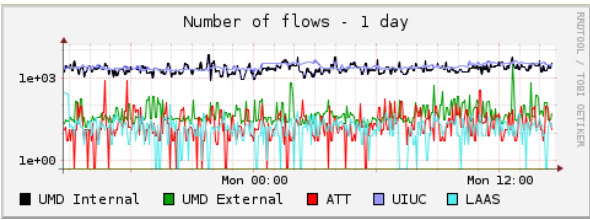


Figure 6: Number of Flows

The tables in Figure 4 show the top and bottom 10 ports targeted by the attackers during the last 24 hours. For each port, the number of flows and the percentage of the total number of flows are provided. It makes it easy to identify the most *popular* services and to protect the network accordingly. The severity of an attack is not related to the number of flows it will generate. Attacks towards common ports tend to hide smaller attacks against less popular ports. This is why we also decided to display the bottom 10 ports targeted.

Finally, Figure 7 represents a word cloud of the top 20 attackers' IP addresses. The top 20 IP addresses are determined using the number of flows involved in the communications between the attacker and the honeypots. The size of the font displaying the IP address reflects the number of flows generated for that IP address. The same representation is used for the top 20 targeted honeypots. These word clouds are updated every 5 minutes using a 24-hour window. The IP addresses presented in the word clouds are clickable: The user can obtain the lists of honeypots contacted, services and Snort events related to the selected IP address in a separate window. Since the honeypot network often hosts different experiments with different configurations, the port tables and the targeted honeypots make it possible to determine what is attracting the attackers the most.

Figure 7: Attacker word cloud

3.3 Snort Data

Last 10 Snort Events

Timestamp	Signature	Source	Destination
2011-06-06 14:41:53	stream5: Limit on number of overlapping TCP packets reached	xxx.xxx.xxx.xxx	174.77.190.64
2011-06-06 14:41:53	stream5: Bad segment, overlap adjusted size less than/equal 0	xxx.xxx.xxx.xxx	174.77.190.64
2011-06-06 14:41:50	ICMP PING NMAP	94.248.15.15	xxx.xxx.xxx.xxx
2011-06-06 14:41:45	stream5: Limit on number of overlapping TCP packets reached	xxx.xxx.xxx.xxx	190.11.17.22
2011-06-06 14:41:45	stream5: Bad segment, overlap adjusted size less than/equal 0	xxx.xxx.xxx.xxx	190.11.17.22
2011-06-06 14:41:45	stream5: Limit on number of consecutive small segments reached	xxx.xxx.xxx.xxx	190.11.17.22
2011-06-06 14:41:44	stream5: Limit on number of overlapping TCP packets reached	xxx.xxx.xxx.xxx	78.187.81.52
2011-06-06 14:41:44	stream5: Bad segment, overlap adjusted size less than/equal 0	xxx.xxx.xxx.xxx	78.187.81.52
2011-06-06 14:41:44	stream5: Limit on number of consecutive small segments reached	xxx.xxx.xxx.xxx	78.187.81.52
2011-06-06 14:41:43	stream5: Limit on number of overlapping TCP packets reached	xxx.xxx.xxx.xxx	94.97.113.112

Figure 8: Last 10 Snort events table

The Snort section presents information about the Snort alerts.

Figure 8 shows a table of the last 10 Snort events collected on the honeypot network. This table allows honeypot administrators to immediately identify attacks generating high volumes of traffic. For example, a brute-force attack against a Microsoft SQL server will generate a spike in the traffic curves and the corresponding events will appear immediately in this table.

The graph in Figure 9 provides a trend in the number of Snort events recorded the current day, the past few days, and the past few weeks.

Figure 10 shows the top and bottom 10 Snort signatures tables. The tables provide the signature name, the number of events for each signature and the percentage. Large scale attacks such as port scanning or brute-force attacks may generate several events. As a consequence, smaller but still important attacks may not appear in the top 10 signatures. This is why the bottom 10 Snort signatures are also provided. As an example, consider the snort signature *SHELLCODE NOOP* shown in the Bottom 10 Snort events of Figure 10. This signature indi-

Top 10 Snort Events (last 24 hours)		
Signature	**Number**	**%**
snort: "SQL sa brute force failed login unicode attempt"	56932	54.55%
"MS-SQL SA brute force login attempt TDS v7/8"	23851	22.85%
stream5: Bad segment, overlap adjusted size less than/equal 0	8357	8.01%
stream5: Limit on number of overlapping TCP packets reached	7805	7.48%
stream5: Limit on number of consecutive small segments reached	2325	2.23%
MISC MS Terminal server request	1519	1.46%
ICMP PING NMAP	1271	1.22%
ssh: Protocol mismatch	801	0.77%
stream5: Data sent on stream not accepting data	308	0.3%
stream5: Packet missing timestamp	305	0.29%
More...		

Bottom 10 Snort Events (last 24 hours)		
Signature	**Number**	**%**
snort: "SPECIFIC-THREATS ASN.1 constructed bit string"	1	0%
WEB-IIS WEBDAV nessus safe scan attempt	7	0.01%
ICMP Source Quench	7	0.01%
ICMP L3retriever Ping	15	0.01%
snort: "SHELLCODE base64 x86 NOOP"	19	0.02%
ICMP Destination Unreachable (Communication with Destination Host is Administratively Prohibited)	25	0.02%
ftp_pp: Invalid FTP command	28	0.03%
"POLICY RDP attempted Administrator connection request"	29	0.03%
stream5: TCP Timestamp is outside of PAWS window	30	0.03%
ICMP Destination Unreachable (Communication Administratively Prohibited)	34	0.03%
More...		

Figure 10: Top and bottom 10 Snort signatures

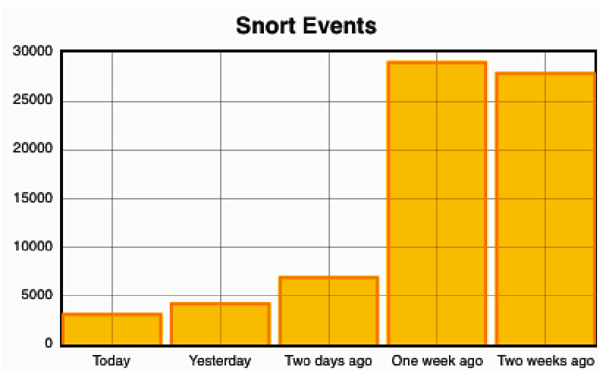


Figure 9: Snort events graph

cates attempts to upload a malicious shellcode.

In the following example, the Snort IDS alerts show a possible injection of malicious code on an emulated Web server:

```
04/15-06:49:15.474819  [**] [1:12799:3] SHELLCODE base64 x86 NOOP [**]
 [Classification: Executable Code was Detected]... {TCP} a.b.c.d:15017 -> W.X.Y.Z.:80
04/15-06:49:15.474819  [**] [1:12802:3] SHELLCODE base64 x86 NOOP [**]
 [Classification: Executable Code was Detected]... {TCP} a.b.c.d:15017 -> W.X.Y.Z.:80
04/15-06:49:15.619028  [**] [1:12800:3] SHELLCODE base64 x86 NOOP [**]
 [Classification: Executable Code was Detected]... {TCP} a.b.c.d:15017 -> W.X.Y.Z.:80
```

The injection was successful and Nepenthes captured and logged the malware submission:

```
[2011-04-15T06:49:19] a.b.c.d-> W.X.Y.Z. ftp://1:1@a.b.c.d:21/Rewetsr.exe
 c511c4f9bdd3bb892e582fbc9a00da9c
```

4 Case Study

This section details the UMD honeynet, the honeypot network deployed at the University of Maryland and also describes how DarkNOC is used to operate and maintain this particular network.

4.1 UMD Honeynet

4.1.1 Introduction

The honeypot network hosted at the University of Maryland was initially built in 2004 with unused IP addresses of the campus network. More recently, other organizations joined the initiative: AT&T Labs, the University of Illinois at Urbana Champaign, and the Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) in Toulouse, France. Each of these organizations contributes to the UMD honeynet by providing ranges of public IP addresses.

The objective of the UMD honeynet is to provide the infrastructure to support honeypot-based experiments. The network features a centralized data collection and guarantees a realistic but controlled and flexible environment to safely deploy experiments. The advantages of the present architecture are multiple:

- A single gateway collects and stores the stores Snort events, flow data and network traffic, providing visibility across the full range of exposed networks.
- The experiments are easy to deploy without the need to create tunnels or to setup specific network configurations.
- The UMD honeynet is scalable, new organizations can join the project by providing range of IP addresses.

4.1.2 Architecture

Figure 11 shows the current architecture of the UMD honeynet and the different institutions involved in the project. A tunneling program called Honeymole[3] redi-

[3] http://www.honeynet.org.pt/index.php/HoneyMole

rects silently the traffic from the different organizations to the UMD honeynet.

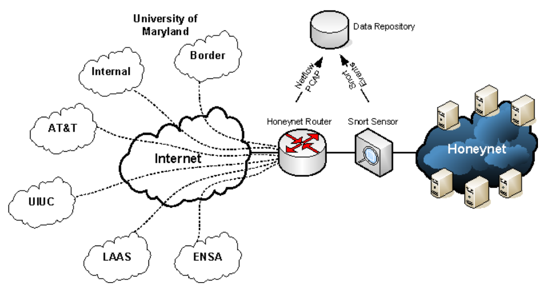


Figure 11: UMD honeynet architecture

The complexity of managing and monitoring such a network was the primary motivation for the development of DarkNOC. This section will discuss the application of the tool to that problem.

4.2 UMD DarkNOC Implementation

4.2.1 Subnet Status

The subnet status section is specific to the UMD honeynet. Each organization involved in the UMD honeynet provides one or more ranges of IP addresses called subnets. For example, the University of Maryland provides two distinct subnets: a subnet of the campus internal network and a subnet at the border network. The failure of a Honeymole tunnel is a significant event for the network, as it implies loss of an entire subnet; the subnet status display allows a manager to quickly assess the status of the tunnels and act on any issues.

Each subnet hosts a low interaction honeypot run by Nepenthes to collect malware. Depending on the network configuration, a Honeymole tunnel may be established to redirect the traffic to Maryland. DarkNOC monitors the quantity of malware collected, the status of the Honeymole tunnels, and the status of the low interaction honeypots.

4.2.2 Compromised Honeypots Detection

Some experiments deployed on the UMD honeynet may present significant risks. In the likely event of a honeypot being compromised, the attacker may use the machine to attack other hosts on the Internet. These attacks are generally easily detectable: Figure 12 shows that the volume of outgoing traffic is substantially greater than the incoming traffic. In this case, a honeypot was used as a proxy server.

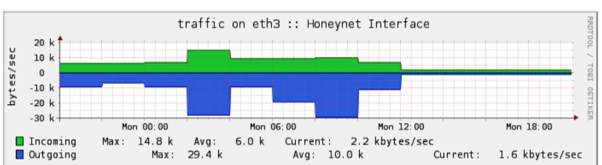


Figure 12: Network traffic (04/18/2011)

4.2.3 Traffic Anomaly Detection

A current experiment uses a known-vulnerable SSH server running on about 80 IP addresses of the Internet subnet provided by the University of Maryland. The DarkNOC's summaries proved useful in analyzing an attack on this configuration of the network which occurred on June 3, 2011.

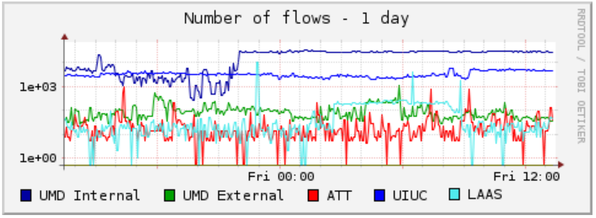


Figure 13: 06/03/2011, number of Flows

1. Figure 13 shows an increase in the number of flows just before midnight on Thursday night.
2. The number of attackers presented in Figure 14 remains relatively steady. This suggests that a fixed set of attackers is generating a large volume of traffic.
3. Figure 15 shows that port 22 is very active. As SSH sessions do not usually generate many flows, we can assume that the attacker is using a bruteforce attack against several IP addresses hosted within the UMD honeynet.
4. The word cloud of the honeypots targeted showed that the IP addresses of this specific SSH experiment were targeted.

DarkNOC provided several indications on the nature of the attack responsible for the spike in traffic network and flows. That night, the health monitoring system of the experiment reported several times that the machine was overloaded and the SSH server failed.

4.2.4 Using Honeypots as a Security Tool

Compromised Hosts Detection
The network traffic observed within an honeypot network is considered malicious. A healthy host would

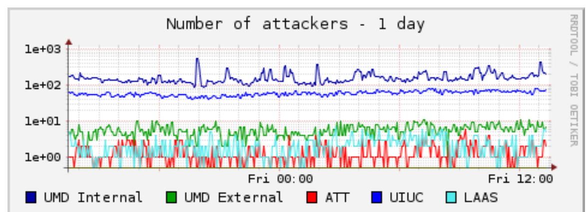


Figure 14: 06/03/2011, number of attackers

Top 10 Ports (last 24 hours)		
Port Number	Number	%
TCP / 22	240597	57.49%
TCP / 27977	20000	4.78%
UDP / 42332	18247	4.36%
TCP / 80	11248	2.69%
TCP / 1433	8464	2.02%
TCP / 443	5552	1.33%
UDP / 19756	5550	1.33%
TCP / 3306	5033	1.2%
TCP / 3389	4513	1.08%
ICMP / 8.0	4286	1.02%

Figure 15: 06/03/2011, top 10 destination ports

not normally communicate with the honeypots. We can therefore use the UMD honeynet to detect compromised hosts on the Maryland campus network. We assume that if a computer on campus appears in the flow data, that means the host is compomised. The alerting module queries the flow data to identify these hosts. This method is efficient at detecting scanners: the use of subnets from both local and remote sites means that a scanner is likely to eventually visit the UMD honeynet whether its probes are directed locally or at the Internet.

When a compromised machine is detected, the alerting module analyzes the event and generates an email that is sent to the IT Security Officer for further analysis. Figure 16 is an example of such a report. For each host, the number of flows, packets and bytes are provided. The report is also available on the Web interface of DarkNOC, it is possible to vizualize the flows associated with the alert. This technique helps to identify compromised hosts and misconfiguration as well. When this alerting system was first launched, the IT team figured that even if a host was tagged as blocked in their systems the compromised host was still able to communicate on the network and to continue its malicious activity. The analysis is performed every 12 hours and each participating organization gets notified of the eventual compromises of their systems. The choice of running the analysis at this frequency was chosen based on the feedback provided by the security team of the University of Maryland. The team wanted to receive a report early in the morning and

```
--------------- Analysis Report ---------------
Flow Time Window: 2011/06/06.06:00:00-2011/06/06.18:00:01
Number of hosts detected: 3
To access the online version of the report:
  https://xxx.xxx.xxx.xxx/darknoc/alert_hosts.php?report=263

xxx.xxx.xxx.xxx (X.umd.edu)
 - Number of flows: 1
 - Number of packets: 1
 - Number of bytes: 51
To visualize the flows:
   https://xxx.xxx.xxx.xxx/darknoc/alert_hosts.php?id=1124

yyy.yyy.yyy.yyy (Y.umd.edu)
 - Number of flows: 10
 - Number of packets: 10
 - Number of bytes: 1915
To visualize the flows:
  https://xxx.xxx.xxx.xxx/darknoc/alert_hosts.php?id=1125

zzz.zzz.zzz.zzz (Z.umd.edu)
 - Number of flows: 10
 - Number of packets: 10
 - Number of bytes: 1915
To visualize the flows:
  https://xxx.xxx.xxx.xxx/darknoc/alert_hosts.php?id=1126
```

Figure 16: Alerting module report

right after business hours.

Security Profiling

Honeypots can provide relevant information regarding attackers and their techniques to compromise a computer. DarkNOC brings together enough information from different datasets to establish a security profile of a network. This profile includes the services targeted, the number of malware uploaded and the types of attacks. The objective is to help the security officers and network administrators to understand where to focus their efforts and to identify weaknesses and misconfigurations. DarkNOC can also be used to evaluate the performance of the security policy in place. The attacks detected and the malware uploaded on the honeypots are good indicators of the efficiency of an IPS device.

Attack techniques are constantly evolving as new vulnerabilities are discovered regularly. The honeypots can help to identify the current trends and to update the security policy accordingly.

5 Related Work

Lance Spitzner defines honeypots as a security tool *whose value lies in being probed, attacked, or compromised* [21]. In other words these are highly monitoring computer systems meant to attract hackers, analyze their modus operandi and profile them [19]. Placed in production environments, honeypots take an active part in the security of a network by providing information on attackers and attacks' patterns. Niels Provos introduces two types of honeypots [18]: high interaction honeypots

that involve the deployment of real operating systems on real or virtual machines, and low interaction honeypots that are computer software emulating operating systems and services.

Companies and researchers currently deploy honeypots networks at different scales. Also known as honeynets, these honeypots networks can be limited to few IP addresses on the local network or distributed systems in several locations such as the Leurre.com project [16], the Internet Motion Sensor [4], SGNET [13] or the honeynet initiative from CAIDA [23].

Levine et al. demonstrated the usefulness of deploying honeypots accross large enterprise networks [14]. In their study, Snort [20] was used to detect compromised computers accross Georgia Tech network. In DarkNOC a similar detection has been made possible by using the flow data. We assume that any traffic seen on the honeypot network is malicious.

The visualization and data analysis of malicious network activity has been the focus of a variety of commercial and open source products. On the commercial side, security companies such as Tenable and Sourcefire offer threat management products that collect logs from multiple devices and generate alerts to inform security analysts about potential intrusions. The main limitation of these solutions with respect to our goal is that they are not tailored to honeypot management and honeynet data collection and so they require additional effort to integrate honeypots in the organization security data analysis suite. Arbor Network is another commercial security vendor that offers a threat management product but the difference with the previous solutions is that they leverage their customer networks to instrument dark IP space at a large scale. As a result, they offer a global view of malicious network activity through their Atlas portal[4], which provides functionalities similar to DarkNOC, with graphs and tables for top attacks, top threat sources and attack trends.

On the open source side, the main honeynet management solution has been Honeywall [8] developed by the Honeynet Project. The Honeywall is a bootable CD-Rom that installs a Linux-based network gateway to manage and control honeypots as well as visualizing and analyzing honeynet logs. Compared to DarkNOC, Honeywall has a more capabilities to actively limit outgoing traffic but it has been designed for small honeypot network. The data processing capabilities of DarkNOC were designed for large scale and multi-site deployments. The objective of the DarkNOC project is to provide a flexible and powerful analysis program. It is adjustable to fit different honeypots configurations. However Honeywall is a all-in-one solution for small scale honeypot networks. It provides routing, capture and analysis capabilities. Integrating Honeywall in an existing large-scale honeypot network is more challenging.

Other open source projects that are not specifically tailored for honeypots include Alienvault [7], Aanval[5], Nfsight [6] and NVisionIP [12]. Alienvault and Aanval are network and system log management solutions that can only process Snort alerts and syslog events while Nfsight works exclusively with Netflow and has been designed for large-scale processing and security visualization of Netflow. NVisionIP processes global network Netflow data to specifically detect attacks and misuses.

Visoottiviseth et al. present a distributed honeypot framework using low interaction honeypots [22] running the honeyd daemon [17]. More specifically, they describe the working of the honeyd logs centralization and their analysis [22]. The framework only works with Honeyd log files. The level of interaction of our framework is also different since we are running low interaction honeypots as well as high interaction honeypots.

6 Future Work

We are working on a number of extensions and improvements on DarkNOC. The first extension will be the addition of a malware section in the user interface. This new section will provide more information about the malware collection including a graph showing the number of uploads per day but also some indications on the methods used to upload the malicious software and its name. The second improvement will be the implementation of the automatic detection of compromised honeypots in the alerting module. This detection will allow DarkNOC to automatically block the outbound traffic of compromised honeypots. Currently, only the detection of compromised non-honeypot hosts of an organization is automated. The graphical user interface of DarkNOC can also be enhanced. There is no option that allows to select and display the activity of a specific period of the day. It would be useful to be able to choose on a graph a particular moment of the day and see the activity at this precise time.

7 Conclusion

In this paper we presented DarkNOC, a honeypot network management and monitoring tool. DarkNOC provides a summary of the activity of the honeypots in the network. This summary is generated from different sources of data including Netflow, malware collected by the Nepenthes low interaction honeypots and attacks detected by the Snort intrusion detection system. Brought

[4]http://atlas.arbor.net

[5]http://www.aanval.com/

together, these data sources provide important resources to help network administrators, security teams, and security researchers understand attacks and protect systems. DarkNOC can be used the detect traffic anomalies and identify interesting case study for research purposes. Since it is important to detect quickly any compromised honeypots in the honeynet, DarkNOC provides administrators of these networks information regarding the health of the systems. Security teams may find a particular interest in DarkNOC since it can be used to detect compromised honeypots as well as compromised hosts on their non-honeypots networks. To sum up an organization using DarkNOC can have a better understanding of:

- the most targeted systems,
- the attackers, the attacks and their origin,

but also, DarkNOC helps:

- to obtain an overview of Honeynets activity,
- to identify security tools and devices misconfiguration.

Acknowledgement

The authors thank the Office for Information Technology at the University of Maryland. In particular we thank Gerry Sneeringer and his team for allowing the deployment of the UMD honeynet, providing feedback on DarkNOC and investigating the compromises detected by the application.

References

[1] S. Almotairi, A. Clark, G. Mohay, and J. Zimmermann. Characterization of attackers' activities in honeypot traffic using principal component analysis. In *Proceedings of the 2008 IFIP International Conference on Network and Parallel Computing*, pages 147–154, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Paul Bacher, Thorsten Holz, Markus Kotter, and Georg Wicherski. Know Your Enemy: Tracking Botnets (using honeynets to learn more about bots). Technical report, The Honeynet Project, August 2008.

[3] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *Proceedings of RAID'2006*, pages 165–184, 2006.

[4] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The internet motion sensor: A distributed blackhole monitoring system. In *In Proceedings of Network and Distributed System Security Symposium (NDSS 05*, pages 167–179, 2005.

[5] R. Berthier and M. Cukier. An evaluation of connection characteristics for separating network attacks. *International Journal of Security and Networks*, 4:110–124, February 2009.

[6] R. Berthier, M. Cukier, M. Hiltunen, D. Kormann, G. Vesonder, and D. Sheleheda. Nfsight: netflow-based network awareness tool. In *Proceedings of the 24th USENIX LISA*, 2010.

[7] Jeramiah Bowling. Alienvault: the future of security information management. *Linux J.*, 2010, March 2010.

[8] G. Chamales. The honeywall cd-rom. *Security Privacy, IEEE*, 2(2):77 – 79, mar-apr 2004.

[9] Kevin Curran, Colman Morrissey, Colm Fagan, Colm Murphy, Brian O'Donnell, Gerry Fitzpatrick, and Stephen Condit. A year in the life of the irish honeynet: attacked, probed and bruised but still fighting. *Inf. Knowl. Syst. Manag.*, 4:201–213, December 2004.

[10] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.

[11] Jan Kohlrausch. Experiences with the noah honeynet testbed to detect new internet worms. *IT Security Incident Management and IT Forensics, International Conference on*, 0:13–26, 2009.

[12] Kiran Lakkaraju, William Yurcik, and Adam J. Lee. Nvisionip: netflow visualizations of system state for security situational awareness. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, VizSEC/DMSEC '04, pages 65–72, New York, NY, USA, 2004. ACM.

[13] Corrado Leita and Marc Dacier. Sgnet: A worldwide deployable framework to support the analysis of malware threat models. In *Proceedings of the 2008 Seventh European Dependable Computing Conference*, pages 99–109, Washington, DC, USA, 2008. IEEE Computer Society.

[14] J. Levine, R. Labella, H. Owen, D. Contis, and B. Culver. The Use of Honeynets to Detect Exploited Systems Across Large Enterprise Networks. In *Proceedings of the IEEE Workshop on Information Assurance*, IEEE Systems, Man and Cybernetics Society, pages 92–99, West Point, NY, June 2003.

[15] Mauro, Ro, and Francesca Mazzoni. HoneySpam: Honeypots Fighting Spam at the Source. pages 77–83.

[16] Fabien Pouget, Marc Dacier, and Van Hau Pham. Leurre.com: on the advantages of deploying a large scale distributed honeypot platform. In *ECCE'05, E-Crime and Computer Conference, 29-30th March 2005, Monaco*, 03 2005.

[17] Niels Provos. Honeyd: A Virtual Honeypot Daemon. Technical report, Center for Information Technology Integration, University of Michigan, February 2003.

[18] Niels Provos and Thorsten Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Addison-Wesley Professional, first edition, 2007.

[19] Daniel Ramsbrock, Robin Berthier, and Michel Cukier. Profiling attacker behavior following ssh compromises. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 119–124, Washington, DC, USA, 2007. IEEE Computer Society.

[20] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX LISA*, 1999.

[21] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[22] V. Visoottiviseth, U. Jaralrungroj, E. Phoomrungraungsuk, and P. Kultanon. Distributed honeypot log management and visualization of attacker geographical distribution. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 23 –28, may 2011.

[23] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.*, 39:148–162, October 2005.

[24] Nathalie Weiler. Honeypots for distributed denial of service attacks. In *Proceedings of the 11th IEEE International Workshops on Enabling Technologies: nfrastructure for Collaborative Enterprises*, pages 109–114, Washington, DC, USA, 2002. IEEE Computer Society.

A Cuckoo's Egg in the Malware Nest
On-the-fly Signature-less Malware Analysis, Detection, and Containment for Large Networks

Damiano Bolzoni[1], Christiaan Schade[1] and Sandro Etalle[1,2]

[1]University of Twente, The Netherlands
[2]Eindhoven Technical University, The Netherlands

Abstract

Avatar is a new architecture devised to perform on-the-fly malware analysis and containment on ordinary hosts; that is, on hosts with no special setup. The idea behind Avatar is to inject the suspected malware with a specially crafted piece of software at the moment that it tries to download an executable. The special software can cooperate with a remote analysis engine to determine the main characteristics of the suspected malware, and choose an appropriate containment strategy, which may include process termination, in case the process under analysis turns out to be malicious, or let it continue otherwise. Augmented with additional detection heuristics we present in the paper, Avatar can also perform signature-less malware detection and containment.

Keywords: system security, malware detection and containment

1 Introduction

In the last half-decade, malware has evolved from a "hobby" for bored programmers to a business for cyber-criminals, who infect computer systems on a large scale to carry out illegal activities [20]. *Botnets* are a typical example of such business, and can be exploited to collect financial/sensitive user information. As noticed by Kolbitsch et al. [13] "malicious code, or malware, is one of the most pressing security problems on the Internet". Malware containment has thus become an urgent concern. Recent events, such as the RSA breach back in March 2011 [17], have shown that serious attackers employ *ad hoc* malware in multi-stage attacks to penetrate corporate networks and get hold of business-critical information.

Successful malware containment is based on two activities: *Detection* and *analysis*.

Detection. Concerning detection, the standard mechanisms employed against malware are based on signatures. Antivirus software and intrusion detection systems (both host- and network-based) rely on some sort of byte-matching techniques (either pattern- or hash-based) to detect the presence of malicious programs. To evade signature-based detection, malware writers can and do obfuscate the code using e.g., polymorphism, packing, encryption [4]. The result of the massive application of evasion techniques is that in the past few years the number of unique malware samples, and relative signatures, has increased dramatically. In Section 4 we discuss in more detail some of the latest results in signature-based malware detection.

Analysis. To understand how malware works, and to improve the crafting of detection signatures, researchers have developed several frameworks for automating dynamic malware analysis (e.g., Anubis [1], CWSandbox [7], Malheur [14]). These tools monitor

the behaviour of a malware sample which is being executed in a severely controlled environment, and produce a detailed report of the operations it carries out (e.g., access/modifications to files, network activities, process execution, etc.).

Dynamic malware analysis is undoubtedly effective; however, it requires a specific analysis environment, which cannot be just *any* computer. Moreover – as almost all security techniques – it is not infallible: Among the possible evasion techniques, it is becoming a common practice for malware to check whether the execution takes place in a virtualized environment, which likely indicates the executable is being monitored [2]. Secondly, as reported by Comparetti et al. [6], some malicious behaviors, such as the so called "dormant functionalities", may remain long unobserved, for instance when they depend on circumstances which are hard to guess and to replicate dynamically.

Summarizing, current detection and analysis approaches suffer from the following limitations:

- (Existing) dynamic malware analysis approaches can only perform post-mortem, or offline, analysis of the malware sample, once it has been collected and submitted: Hence they lack the execution context information; moreover, they require specific setups.
- Detection and containment are based on signatures or behavioral models, and are therefore effective only for those samples for which an appropriate signature/model has been developed.
- The most effective approaches rely on the presence of an agent on the end-host to monitor system activities; such extra software component is invasive, might affect system performance, and cause additional burden because system administrators must plan carefully its development and maintenance.

In particular, security analysts do not get the chance to analyze and contain on-the-fly suspicious programs.

One would like to have in addition to standard tools a first line of defense against malware that does not require special settings for the host, nor predeployed signatures. Similarly to what happens with intrusion detection systems, and especially for large corporations, one could think of a security operation center (SOC), where security analysts are able to inspect on-going suspicious behaviours. Thus, automatic analysis tool could be employed to "select" suspicious programs for analysis, which would be then carried out with a mix of automatic and manual inspections.

Contribution In this paper, we present a novel approach to perform *on-the-fly* malware analysis and containment for large networks, without having to deploy any end-host component beforehand. Our architecture, we call it Avatar, relies on the observation that malware distribution is usually done in at least two phases: First the computer is infected with a tiny "spore", then in following phases this spore downloads one or more additional components from, for instance, some earlier compromised web servers. Those components, or "eggs", are used to extend the malware capabilities, e.g., hooking system APIs to grab user passwords, and usually come in the form of executables, or dynamic libraries. By doing so, malware writers can more easily avoid detection.

Our approach is based on the injection of "goodware" in the suspected malware: In the moment that the alleged malware attempts to dowload an egg, we substitute the egg with the goodware, we call it the *cuckoo's egg*[1]. This is an executable that – among other things – can carry out preliminary malware analysis, can terminate the malware or it can simply give the control back to the egg if the suspected malware turns out to be a legitimate program[2]. The current implementation of Avatar is meant to monitor Windows-based systems.

This is done without any special setup in the host

[1]Similarly to the cuckoos that engage in brood parasitism, our goodware is expected to circumvent the malware and take advantage of it for performing the analysis

[2]In some cases it may be illegal to inject in an application software other than the one meant to be downloaded. Avatar is meant to be deployed in corporate networks, where system and network administrators are (usually) allowed to monitor, and limit, users' actions.

that contains the suspected malware, which may be just any computer running any Windows operating system. Indeed, the cuckoo's egg can be generated and inoculated from the firewall, and the analysis can be done on a remote analysis engine to which the cuckoo's egg communicates after it has been injected in the host under analysis.

Our experiments show that this is all possible, and that the cuckoo's egg can, for instance, be designed to inspect the process that executes it after the download, or to send to Avatar's remote analysis engine information regarding the process, such as path on the file system, file handlers, network/registry activities, or even the executable itself. Depending on the current user's permissions, the malware analysis engine can even "order" the cuckoo's egg code to suspend or terminate the process, effectively containing a possible larger infection.

An important side-issue is when should one start being suspicious about a given process. In other words, when should the system suspect that a spore is actually trying to download an egg. For our experiments we have developed a heuristic method which works as follows: Malware is usually programmed to use several different download servers, as servers are often offline/discontinued. In practice, the spore often fails a number of times before succeeding in downloading the egg. Thus, we take into consideration per-host failed TCP connections and failed HTTP requests to identify malware attempts of downloading. A number of failed HTTP requests is a good indication of the presence of malware. Our experiments show that this method is surprisingly effective. However, one can devise other heuristics which may be applicable in other contexts. It is outside of the scope of this paper to make an inventory of such methods.

To the best of our knowledge, this is the first approach which – without the installation of any additional plug-in before hand – allows one to:

- **(analysis)** carry out on-the-fly remote analysis of a suspicious program;
- **(containment)** suspend or terminate the suspicious program directly on the infected host;
- **(detection)** in combination with the heuristics for detecting suspicious downloads, it can identify suspicious malware processes which can be immediately analyzed and contained if required.

We should remark that this is done without using signatures of any kind. Therefore, this approach can be used to detect, analyze and contain also zero-day malware and malware for which there is no signature available yet. For example, one could even think of a "paranoid" mode, in which a cuckoo's egg is shipped for each download of executables regardless the rate of failed connections.

We show that Avatar is effective as a lightweight first line of defense against malware, also allowing to do malware containment on hosts with no specific pre-deployed tools (agent-less). This is a crucial requirement for system administrators of large networks, as it eases the burden required to install additional software to perform an accurate monitoring.

It is important to stress that this approach can be adapted to work with any protocol, in our embodiment we choose HTTP because it is widely used by malware writers. Of course, this approach has limitations, and can be countered to some extent. These aspects are discussed in Section 2.6.

2 Architecture

The architecture of Avatar consists of three main parts. The download detection engine (DDE) is responsible for detecting suspicious attempts to download software components. The Cuckoo's Egg Generator (CEG) is responsible for crafting the special analysis software that will be sent to requesting host. Finally, the Malware Analysis Engine (MAE) is responsible for analysing the information provided by the injected cuckoo's egg and possibly initiate some containment strategies. We now provide a detailed description of each component.

2.1 Download Detection Engine

The download detection engine (DDE) detects (failed) download attempts that *might* be due to malware activity. Strictly speaking, the functioning of the DDE is orthogonal to that of the analysis and

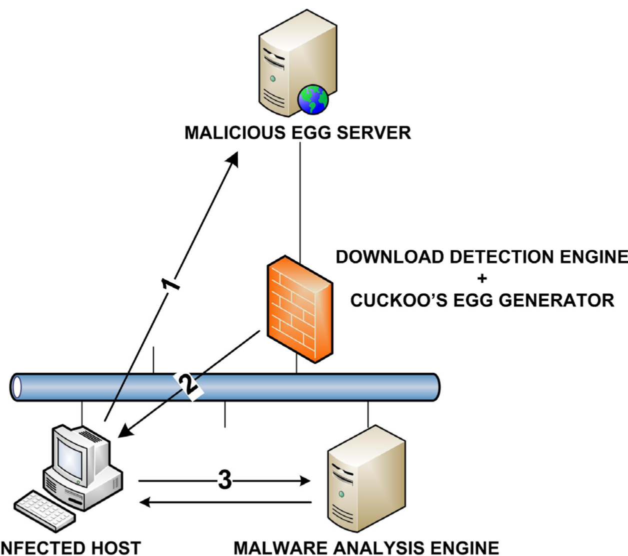


Figure 1: The Avatar architecture.

containment engines of Avatar, which on the other hand, are the core of the system. In fact, Avatar would work just as well also in combination with any other method one could devise to spot out suspicious download attempts. Nevertheless, it is easier to explain the whole architecture by starting from the DDE.

Our DDE is based on the fact that often malware fails a number of time to download eggs. This is due to the fact that download servers are often offline and/or taken down by security officers. In our embodiment the detection engine combines a modified version of the Threshold Random Walk (TRW) algorithm [10]. The engine builds a per-host model of normal usage, which takes into account the number of failed connections, and failed HTTP requests. In the case of malware, the former situation can occur when, e.g., the remote web server has been deactivated, the latter because the malicious content has been removed. As confirmed by our tests (see Section 3.1), these are not infrequent events. The resulting algorithm is simple, albeit effective, and could be easily expanded to include additional sources of information (e.g., DNS queries).

The DDE may be located at the network "border" with the Internet, in order to observe any outgoing connection and the data sent back by the remote host. As we said, while this component plays an important role in our approach, it is not the main driver of our idea. For instance, one could decide to inspect any executable download from the Internet, without the host having to failed a number of connections or HTTP requests before being flagged as suspicious.

The TRW algorithm is devised to detect scanning behaviours originating from a specific host in a monitored network. For each host a detection model is built. The outcome of a connection attempt is either "success" or "failure". After a number of observations of connection attempts for a certain host h, one would like to know if h is a scanner. To make such decision, a sequential hypothesis testing method is used. The basic premise is that there exists a distinct fixed ratio of failed and successful connections, and that this ratio is different when a host is a scanner. Furthermore, for each individual host this ratio value will eventually converge to some upper or lower boundaries, based on whether the host is a scanner or not.

We have adapted the TRW algorithm to take into account also successful and failed HTTP requests:

We currently employ only one model for both TCP connection and HTTP requests.

2.2 Cuckoo's Egg Generator (CEG)

Once the DDE identifies a suspicious download attempt, the CEG generates a specific executable/DLL to be fed back to the suspicious host. We now describe in details the purpose of the cuckoo's egg and its "internals".

2.2.1 The cuckoo's egg

The main goals of the cuckoo's egg are I) to gain as much knowledge as possible about the executing process, that, in case of malware, is usually the process that tried to download the "egg" and received the cuckoo's egg instead of it, and II) to take control over the parent process if necessary. The cuckoo's egg operates in two stages.

First, the cuckoo's egg "inspects" the execution environment. The reason for this is that different operating systems allow processes to execute certain operations with or without high privileges. Therefore, the cuckoo's egg may be allowed to perform only a restricted set of operations. For instance, beginning with Windows Vista, Microsoft includes a User Access Control (UAC) mechanism. The system can be set to notify the user when a process is about to modify some important system settings or execute potentially dangerous operations, so that the user can give explicit authorization. Because we want the cuckoo's egg to be as transparent as possible (for usability reasons), on Vista and later OSes, we cannot use a number of features, such as debugging mode, as these could (possibly) trigger the UAC.

The cuckoo's egg attempts to inject a specifically crafted DLL into its parent process with different access rights: The parent process can restrict the operation set the child process is allowed to perform. The different combinations of access right masks the cuckoo's egg uses are: PROCESS_ALL_ACCESS (highest privileges), TERMINATE_PROCESS | QUERY_INFO | READ, QUERY_INFO | READ and TERMINATE_PROCESS (lowest privileges).

Secondly, the injected DLL extracts, if allowed to, some information from the parent process (depending on the operational mode, see Section 2.4). This information includes: Full path, executable size on disk, DLLs that have been loaded, and information related to the current window attached to the process (if any), such as handle, size and caption text. At this stage, the cuckoo's egg's DLL attempts to determine quickly whether the parent process is malicious or not, and employs initially some heuristics based on the data above. Our experiments show that in most cases, one could tell straight after these heuristic checks whether the parent process is likely to be malware. For instance, a large executable size (more than 5 MB) is a sign of a non-malicious process: Malware writers tend to reduce the size of the "spore" to by-pass more easily anti-malware countermeasures. Similarly to [13], we also whitelist applications that could perform a licit download and later execute the downloaded file (e.g., Internet Explorer, Windows Update). Some limitations apply to these heuristics, and we discuss them in details in Section 2.6. An additional heuristic one might think to apply is the approach presented in [18], based on PE header analysis of suspicious programs.

If the heuristics do not indicate that the process is legitimate, then the information is passed to the MAE (discussed below) for remote analysis. Then – depending on the operational mode set and the user access rights – the cuckoo's egg can I) debug the parent process, II) let it run normally, III) "freeze" it, and, as a very last countermeasure, IV) terminate it.

In the first case, the cuckoo's egg can send back to the highly-instrumented malware analysis engine the debugged instructions. By doing so, we can "reply" on the remote analysis engine any operation and set whether we are debugging a malware process. However, our experiments show that this approach collects very little useful information on the parent process, as the malign process usually executes the egg(s) as the very last step of its run.

In the second and third cases, the cuckoo's egg sends back to Avatar the parent executable, and this is also the reason why we need to collect the parent's full path. By sending the whole executable, we can restart from scratch the process execution within our

monitored environment and off-load a more accurate analysis.

2.2.2 Packaging the cuckoo's egg

Once the DDE notifies the CEG, the latter has to generate a suitable cuckoo's egg for the target and, depending on the operational mode, "attach" the original executable to the cuckoo's egg. We distinguish two cases, depending on whether the original executable is available, because it could be downloaded, or not.

As we mentioned earlier, the original executable requested by the host may not be available. In this case, only the cuckoo's egg is sent back to the target without any further processing. If, on the other hand, the originally requested executable is available, and the operational mode allows to do so, the CEG "forces" first the execution of the cuckoo's egg, and then of the "real" executable. Hence, the main concern when shipping the cuckoo's egg is to preserve the egg's functionalities as much as possible. There are several ways to achieve this, two of which are discussed here, each preserving the functionality in a different way:

- injecting a DLL loader stub through Portable Executable injection;

- shipping a replacement-executable that fetches and executes the egg after the parent process has been analyzed.

In the first case, the Portable Executable (PE) file header of the downloaded egg is altered. The PE format [15] is a file format for executables, object code, and DLLs, used by Windows since early NT versions. When an executable is launched, the system process loader uses the information included in the PE header to carry out operations such as: Filling in-memory data structures, loading required DLLs, and jumping to the entry point of the executable. In this case, the CEG appends the cuckoo's egg to the egg's executable file, next the egg's Entry Point is modified to point to a loader stub that will unpack the engine and write it to a file after which it will be loaded like any regular DLL. This method is rather complex, it presents the disadvantage that it might trigger the antivirus (unless some packing techniques are used) and requires the *LoadLibrary* and *GetProcAddress* offsets to be available in the egg's PE header, which is usually the case, though.

The second method is much simpler, requires no modifications to the egg's executable file, is usually not flagged as malicious by the antivirus and does not make any assumptions on the egg's PE header. A stand-alone cuckoo's egg is sent back and, once the analysis is over, it downloads and executes the "original" egg. The downside to this approach is that any relation that the egg may want to set up with its parent is lost. Moreover, this could significantly slow down the execution, by introducing an additional download latency.

2.3 Malware Analysis Engine

The MAE is the core component of the Avatar architecture. It is responsible for analysing the information sent by the cuckoo's egg. If necessary, it should run the suspected executable in a protected environment. From a functional point of view, it does not differ from other malware analysis tools. Once the sample to analyse is received, it is executed and any operation performed is recorded and logged. The execution report can be then dispatched to a security analyst, who can set a final verdict about the maliciousness of the sample, in case the executed program's nature remains unclear.

The MAE is also used to store information about whitelisted programs, which the cuckoo's egg will consider as non-suspicious. By doing so, we can basically centralize our architecture, making it possible to "update" crucial information about malware in one step.

2.4 Operational modes

As networks, and hosts, require different confidentiality and availability levels, users need to control the way the cuckoo's egg could affect the execution of processes. As in the case of all detection and prevention systems, false positives are always possible, so

one has to find an appropriate compromise between rigorous containment, at the risk of terminating a legitimate process, and less drastic measures. In our embodiment, we have implemented three basic operational modes.

Transparent mode When in this mode, the DDE notifies the CEG about the failed attempts to pull down some files from an external server. The CEG then waits for the file to be actually downloaded, and verifies it is an executable. If so, the CEG crafts a cuckoo's egg with the original file appended. Once the execution of the cuckoo's egg is over, the original file is automatically executed. The cuckoo's egg sends back to the MAE a copy of the parent executing it for analysis. No further action is possible on the suspicious host, as the cuckoo's egg releases the parent process' executable. This mode does not interfere with regular operations of the suspicious host, as the original requested file is executed.

Semi-transparent mode This mode differs from the transparent mode as follows. The original file is downloaded and attached to the cuckoo's egg. However, when the cuckoo's egg is executed, it freezes the parent process. Then, the cuckoo's egg runs the heuristics checks and might decides to "release" its parent process immediately. If the heuristics checks cannot clearly determine the nature of the parent process, the cuckoo's egg ships a copy of the parent process' executable to the MAE. Then, it waits for further commands from the MAE. Further commands may include the termination or release of the process. This mode might interfere with the regular operations of the suspicious host, as the parent process is frozen while the analysis is in progress.

Non-transparent mode When in this mode, the CEG is notified about the failed downloads, but, provided the requested filename points to an executable, does not wait for the original file to be successfully downloaded. Instead, it immediately ships a cuckoo's egg. Based on the heuristic checks, the cuckoo's egg might send back to the MAE a copy of parent executable, and waits for further commands. This mode heavily interfere with the regular host operations, as the requested file is not executed.

2.5 Implementation

To carry out our experiments we have implemented a proof of concept version of Avatar. The three main components of Avatar can be placed at different locations on the network. However, in our experiments we have coupled the DDE and the CEG together into a single host. The reason for this is that the DDE and CEG must exchange information about failed downloads, and the CEG must craft and supply the cuckoo's egg in a timely manner. The deployment of these two components on physically separated systems might introduce delays that could impact the analysis.

In practice, to allow a transparent deployment that does not require any reconfiguration at host side, we employed a single Linux box with built-in firewall and web proxy. The firewall transparently redirects the outgoing traffic directed to common HTTP ports (TCP ports 80 and 8080) through the web proxy, which can inspect both request and reply. Thus, no re-configuration of client hosts is required. As firewall, we use Netfilter, the Linux sub-component in charge of managing network communications. Netfilter offers the possibility to insert specific "hooks" in its packet process workflow, so that it is possible to inspect, and even modify, on-the-fly any packet passing by. To inspect of HTTP traffic, we set up a web proxy based on Apache. Apache supports modules for adding new functionalities, and we have developed a new module to inspect requests and their content. Internally, the module maintains a table that contains statistics about internal hosts and their connection/request failure rates. The module also inspects the replies sent back by the remote (web) server.

When the same host performs several failed connections in a given timeframe, or requests to pull down some file(s) do not end successfully, the Apache module marks that host as suspicious. Depending on the operational mode, the module will either wait until a request is successful, and then ship back a crafted cuckoo's egg together with the original file, or it will immediately ship back a cuckoo's egg (provided the

request points to an executable filename).

If the requested file is eventually downloaded, the module proceeds with some sanity checks and verifies that the downloaded file is actually an executable. In case of a positive match, the executable is stored and the cuckoo's egg crafted. To craft the cuckoo's egg and append the requested file, we implement the first method presented in Section 2.2.2 (PE header injection), to avoid download latency.

The analysis engine is implemented as a Windows kernel driver. In order to monitor malware activities, the driver hooks some APIs functions, and exploits the capabilities offered by the latest Windows OSes, which provide built-in sub-systems for third-parties antivirus and firewall software. These interfaces allow one to detect changes in the file system, the system registry, monitor network connections, etc.

Technically speaking, the MAE resides on a real system behind a firewall, in order to prevent any outgoing connection that could be initiated by the malware once it is activated. The MAE does not run on any virtualized environment, to avoid possible built-in anti-analysis capabilities inside the malware. This choice has the disadvantage of requiring a roll back to the original status after each analysis. We do not see this as a serious limitation because our current goal is not to speed up malware analysis, which would require several concurrent systems. Nevertheless, the kernel driver can be deployed in a virtualized environment too.

The cuckoo's egg communicates with the analysis engine through encrypted network sockets. Encryption is used to avoid leaks of any possible sensible information, e.g., a memory dump, over the network, and to prevent the spore from tapping our communications.

2.6 Limitations and evasion of Avatar

In this section we discuss the limitations of our approach.

Limitations of the CEG When crafting the cuckoo's egg, the original requested file can be attached to it. This process could break self-extracting archives, which verify the file integrity before inflating the content.

Evading the DDE Our approach works by first detecting (failed) attempts to download additional components. If malware evades this detection phase, then Avatar cannot ship the cuckoo's egg. To avoid detection, malware could initiate connections at a very low rate, as part of our detection relies on high rate of failed connections. Encrypting connections could be also a countermeasure against inspection.

Evading the CEG Another possible way of evading Avatar is by using some sort of verification mechanism of the downloaded components. Encryption and hashing could be employed to detect a mismatch with the expected file. For instance, by compressing the executable and protecting the archive with a password. Because the sanity check performed on the downloaded file can be solely based on the magic numbers only, a malware writer could hide the executable within a different file type and change the file header at run-time, once downloaded.

Evading the cuckoo's egg Because the cuckoo's egg employs heuristics to decide whether to continue the analysis or to send back to the instrumented host the parent executable for analysis, malware could take some countermeasures to evade the heuristics checks. For instance, since Windows 2000, a process can execute instructions within the context of another process by using the *CreateRemoteThread* API function (a similar function allows the injection of DLLs). Thus, malware could inject arbitrary malicious instructions in the context of an accessible whitelisted process, e.g., Internet Explorer, which is usually executed with the same access rights the malware has, to evade some checks performed by the cuckoo's egg[3].

[3] It is worth noting that the very same technique could be used to evade approaches like the one presented in [13], which relies on the fact that some processes can be whitelisted before hand to avoid false alerts.

Possible Solutions Although we acknowledge that it is possible to devise malware with anti-analysis features tailored for our approach, we did not observe any of those during our experiments. Moreover, the use of encryption of hashing for file verification would likely slow down the malware spread, as either "updated" versions would fail the check or researchers could reverse engineer some malware samples and identify the encryption key/password of the mechanism.

By the way, we think that malware writers might be reluctant in adding a verification step to the malware, as it might simplify the work of signature-based detection system. In the moment that the malware is analyzed the key used for encryption would certainly be identified, and this could be used to craft an effective signature for detecting it.

A possible solution to the evasion of the cuckoo's egg would be to add a comparison of the executable on the disk with the memory image, and pinpoint possible later-added instructions. However, this would require also to inspect DLLs, and the task could easily become infeasible (let alone not being bullet-proof). We plan to address in future work this issue.

3 Benchmarks

To validate the effectiveness of our approach, we use two different datasets. The first data set, referred to as DS_A is available on request from the team that built Malheur. It contains a large collection of malware samples that could be used for malicious purposes. In practice, the data set is a collection of samples submitted in a period of eight consecutive days in 2009. Each sample has been analyzed by CWSandbox and the related report is included together with the original sample. This data set is used to test the basic idea of our approach, that malware will execute an arbitrary generated "egg".

Our second dataset, DS_B, is a collection of malware samples found in the wild. For some samples, no report was available beforehand (meaning they were brand new or modification of known malware samples). Hence, we had to submit the sample to either Anubis or CWSandbox to learn whether the sample was actually malware and downloaded some extra components. With this data set we want to test in particular the effectiveness of the devised heuristics for triggering instrumented analysis of the suspicious process.

3.1 Tests with DS_A

This dataset is an extensive collection of malware samples. They belong to different malware families and are all unique, meaning that some sort of polymorphism/code reordering has been applied.

However, not every sample downloads extra components, and among those which perform download activities, a large part cannot work properly these days. This is due to the fact that, before downloading the extra executables, the malware sample attempts to download some configuration files, which are not longer available. We select only working samples that download additional components, and up to 10 maximum samples per family (in total 75 samples).

To perform the experiment, we set up a client host running Windows XP SP3, as some malware samples suddenly crash when executed under more recent OSes[4], like Windows 7. No extra user activity is simulated. For the DDE, we use the following settings: 5 failed connection/download attempts in 1 minute indicate a possible malicious program. The operational mode for this dataset is set to transparent mode. Table 1 summarizes our findings.

Discussion Tests on DS_A show the effectiveness of our approach. However, we have observed that for few samples and for a certain malware family in particular, the cuckoo's egg is not actually executed. There are two distinct reasons for it. In the case of random samples, once the cuckoo's egg injects its crafted DLL the parent process crashes. In the case of the "Killav" malware family, the malware sample relies on the user to actually execute the download file(s). In all the other cases, there is no check run by the malware whether the downloaded file is actually a "legitimate" malicious component. This enforce our assumption that malware

[4] We investigated this issue and found some incompatibles among installed and expected system libraries.

Malware family	**# of samples**	**# of samples marked as malicious by the DDE**	**# samples that executed the cuckoo's egg**
Agent	9	9	9
Adload	8	6	6
Banload	3	2	2
Chifrax	2	2	2
FraudLoad	8	5	4
Genome	4	4	4
Geral	9	8	8
Killav	6	5	0*
Krap	6	4	4
NothingFound	10	10	3
Xorer	7	6	4

Table 1: Actual samples used in our tests with dataset DS_A, samples flagged as malicious by the DDE and that executed the shipped cuckoo's egg. The * marks a family of malware that actually downloads the cuckoo's egg, but does not automatically execute it (and leaves this to the user). In most cases, the DDE detects failed download attempts, and the cuckoo's egg is executed right away by the malicious sample, without any integrity check.

writers do not currently protect their programs with encryption/hashing mechanisms.

For the "NothingFound" family, whose name might refer to the fact that the submitted sample has not beed identified as malicious by CWSandbox, we have to report that the cuckoo's egg has been actually executed most times.

3.2 Test with DS_B

This dataset is used to tests how our approach performs with (supposedly) brand new malware. Samples have been collected in March 2011, and most of them would have not been detected by several antivirus software at the time of collection (we processed each sample through the VirusTotal [23] web site). We have a total of 30 malware samples from this dataset, which downloads extra malware components. For this set of tests, we also simulate regular user activities such as browsing and downloading, with 30 different software, ranging from web browser to crawlers. Because the downloading program might not execute the cuckoo's egg, we automate its execution and set the parent process to be the downloading program. For the DDE, we use the following stricter settings: 3 failed connection/download attempts in 1 minute will indicate a possible malicious program.

To perform the experiment, similarly to the tests with DS_A, we set up a client host running Windows XP SP3. The operational mode for this dataset is set to semi-transparent mode. By doing so, we test at the same time how efficient heuristics are in detecting malware programs. Because some goodware programs that the heuristics might send to the MAE for analysis could rely on the presence of certain system libraries, for this experiment the MAE is running on a mirror copy of the attacked system. When samples are sent to the MAE, we set a maximum amount of waiting time without operation performed of 3 minutes: By doing so we avoid false positives in case of goodware, but might introduce false negatives in case of malware. Table 2 summarizes our findings.

Discussion This second round of tests confirms that even the latest malware code is still "vulnerable" to the injection of our cuckoo's egg. Most samples have been correctly identified by the DDE, and only 2 samples have been missed. These samples have stopped their download attempts just after a

		# of samples
Malware	Correctly identified by the DDE	28/30
	That executed the cuckoo's egg	27/30 (27/28)
	Correctly identified as malware by heuristics	13/30 (13/27)
	Erroneously identified as goodware by heuristics	2/30 (2/27)
	Sent to the MAE for analysis	12/30 (12/27)
Goodware	Erroneously identified by the DDE	10/30
	Correctly identified as goodware by heuristics	6/30 (6/10)
	Erroneously identified as malware by heuristics	2/30 (2/10)
	Sent to the MAE for analysis	2/30 (2/10)

Table 2: Results for tests with dataset DS_B (in the third column we report partial results in brackets). Almost any malicious download attempt has been detected by the DDE, which shipped the cuckoo's egg. The heuristics identified malware samples in almost half cases, and mistakenly flagged as goodware malicious samples only in a couple of cases. The false positive rate for the DDE is around 30%, and around 20% for the heuristics (when considering the cases in which the cuckoo's egg was shipped).

few tries. The DDE also mistakenly detects as malware some regular programs. Actually this was an expected behaviour, as we set strict values for the DDE. Only one program did not execute the shipped cuckoo's egg, due to a crash at the moment of injection. We experienced the same problem for several samples from DS_A, and our investigations show that the malware was not fully compatible with the installed set of libraries, and therefore would have crashed anyway.

The heuristics prove to be an effective way to reduce the workload of the analysis engine (and thus the manual workload for security analysts), when working in combination with the DDE. Less than 50% of analysed malware samples have been sent to the MAE for analysis, and more than 50% of samples have been immediately identified as malware. In the case of non-malware samples, 20% of analysed samples had to be sent to the MAE for further analysis while 60% has been identified as non-malware without further analysis. For each successful execution, the cuckoo's egg would have been allowed to terminate the parent process, effectively containing a possible infection.

3.3 System overhead

Because the Avatar architecture performs an on-the-fly analysis, and eventually detection, the program execution is slowed down by a certain overhead. Depending on the operational mode, this overhead can range between milliseconds and minutes. A typical example of the former case is transparent mode, when the heuristics flag the downloading program as actual goodware. In non-transparent mode, if the heuristics fail to determine the actual nature of the analysed program, the MAE is activated for further inspection (thus, slowing down the overall analysis time). All our benchmarks have been performed on a 1.8Ghz Dual Core machine (target) and 3.0Ghz Quad Core machine (MAE). Table 3 reports overhead values for different scenarios we observed during our benchmarks (overhead refers to the execution time of the cuckoo's egg and, possibly, of the MAE).

4 Related work

In this section we discuss related works. As malware has become a serious security threat, a good deal of work exists that discusses techniques to analyse and detect malicious code.

4.1 Malware Analysis

Sidiroglou and Keromytis [19] present an architecture to detect and capture potential malware infection vectors by using a collection of heterogenous detec-

Dataset	Min overhead	Avg overhead	Max overhead
DS_A	17ms	21ms	30ms
DS_B (malware samples)	19ms	61s	181s
DS_B (goodware samples)	16ms	29s	181s

Table 3: Overhead time values for DS_A and DS_B. When heuristics successfully identify the analysed sample, the overhead can be as low as 16ms. The maximum overhead value depends on the MAE analysis.

tion engines. Engines range from host-based sensors monitoring the behaviour of applications and OSes to honeypots that simulate possible target applications. Each time a potential malware vector, e.g., a byte stream, is detected, it is copied and forwarded to a sandboxed environment, which runs some instances of the applications one wants to protect (e.g., the Apache web server) and a number of tools to verify the potential maliciousness of the input. The authors provide several strategies for fixing, among others, buffer overflow vulnerabilities "on-the-fly". Despite the fact that authors do not provide any implementation of their architecture, there are several similarities with our approach. Once the cuckoo's egg is being executed, the suspicious program is copied and forwarded to a sandboxed environment for dynamic analysis. The main difference lies in the way we inspect the suspicious program, by crafting the cuckoo's egg and sending it together with the original requested file.

Anubis [3] and CWSandbox [24] are two prominent architectures for dynamic malware analysis. In particular, Anubis can aggregate malware samples that present a similar behaviour into "clusters". That is, although samples' diversity is high (Anubis has analyzed more than 1 million of unique malware samples so far), there are nearly 100.000 malware "families".

4.2 Malware Detection

A number of heterogeneous techniques have been presented to detect malware.

Host-based Techniques Host-based techniques were the first to be used to detect and stop malware (think of antivirus software). Their main advantage is that they can detect malware even before it is actually executed. Approaches range from simple byte-pattern matching, which scans a file for known malicious strings or instructions [21], to model checking [12] and compiler verification [5]. Unfortunately, such (static) techniques can be evaded using packers and polymorphism.

In an effort to overcome typical limitations of matching-based approaches, Kolbitsch et al. [13] introduced a new concept of signature based on fine-grained models. Fine-grained models are graphs representing system calls invocation order (and other additional information) to match the characteristic behavior of a given malware program. The model generation is off-loaded onto a dynamic malware analysis tool (i.e., Anubis). This approach allows the detection of unknown malware samples too, provided the "family" has been analyzed before.

Network-based Techniques Regarding specific network-based techniques, several approaches leverage information extracted by analyzing network traffic [8, 9, 11, 16].

BotMiner [8] combines a number of different traffic monitoring tools to extracts network communication patterns and their content. Typical information that BotMiner takes into consideration are vertical and horizontal scans, exploit attempts, DNS queries, downloads of binaries. Then, BotMiner clusters hosts with a similar behavior and attempts to detect botnet nodes. Although network-based approaches could allow, in theory, to perform on-the-fly detection, this is hard to realize because they miss the activity performed by malware on the host.

Techniques based on Data Mining Several researchers address the detection of malware by using data mining techniques, in a effort to detect a higher

number of malware samples that are simply a variant of already known samples.

Tabish et al. [22] notice that most of current malware samples that are daily submitted for analysis are not brand new. Commonly, malware writers employ techniques such as repacking to "obfuscate" malware content and thus defeating approaches based on content matching, e.g., antivirus software. The authors devise an approach based on extracting statistical and information-theoretic features from file blocks. A block is a fixed-sized chunk of byte-level contents of a given file. More than 50 distinct features are extracted, and then analyzed using mathematical distance functions that are common in the data mining field (e.g., the Manhattan and Chebyshev distances). The approach gives in general good results, but requires the analysis of several "good" file samples, e.g., executables, PDF documents, etc., to detect malicious files.

5 Conclusion

In this paper we present Avatar, a new lightweight architecture for on-the-fly, signature-less malware analysis, containment and detection for large networks.

Avatar does not require any special setup or software on the infected hosts. This is because the analysis is not done on the allegedly infected host, but it is carried out on a remote system, which communicates with the (allegedly) infected host through the cuckoo's egg. The cuckoo's egg provides also containment functionalities. In fact, Avatar's architecture is completely centralized. This allows one to deploy it in any environment (like a corporate network) where the firewall can be modified to provide the needed facilities for the interception of suspicious downloads and the injection of the cuckoo's egg. Basically, Avatar can be deployed in most work environments with very little effort. An additional advantage of a centralized architecture is that the updates in the analysis engine affect only one machine, as opposed to what happens e.g., with antivirus software, where all hosts have to be updated.

An interesting aspect of Avatar's architecture is that it can avoid some evasion techniques used by malware; as we mentioned before, modern malware can check whether it is running in a sandboxed environment. Since our architecture does not deploy any extra tool, not even at kernel level, before hand, the malware has little way of detecting that it is under analysis.

The detection in Avatar is necessarily based on heuristics, and is thus fallible. This however allows the detection of malware for which there is no signature available yet. On the other hand, since the heuristics-based detection phase is always followed by an analysis phase before proceeding to the containment, the risk of having false positives in the detection phase is heavily mitigated by the fact that if the analysis phases determines that the suspected malware is actually a legitimate program, the cuckoo's egg can simply "release" it and allow it to continue.

Our experiments show that our approach is effective in detecting and containing malware, even unknown malicious code. We believe that Avatar can be the basis of an effective lightweight first line of defense against malware.

References

[1] Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org`.

[2] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS '10: Proc. 17th Network and Distributed System Security Symposium*, 2010.

[3] U. Bayer, A. Moser, C. Krugel, and E. Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[4] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA '04: Proc. ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44. ACM Press, 2004.

[5] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-Aware Malware Detection. In *S&P '05: Proc. 25th IEEE Sym-*

posium on Security and Privacy, pages 32–46. IEEE Computer Society, 2005.

[6] P. Milani Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *S&P '10: Proc. 31th IEEE Symposium on Security and Privacy*, page TO APPEAR. IEEE Computer Society Press, 2010.

[7] CWSandbox. `http://www.cwsandbox.org`.

[8] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *USENIX Security '08: Proc. 17th Usenix Security Symposium*. USENIX Association, 2008.

[9] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: detecting malware infection through IDS-driven dialog correlation. In *USENIX Security '07: Proc. 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16. USENIX Association, 2007.

[10] J. Jung, V. Paxson, A.W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *S&P '04: Proc. 25th IEEE Symposium on Security and Privacy*, pages 211–225. IEEE Computer Society Press, 2004.

[11] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proc. 13th USENIX Security Symposium*, pages 271–286. USENIX Association, 2004.

[12] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting Malicious Code by Model Checking. In *DIMVA '05: Proc. 2nd International Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, volume 3548 of *LNCS*, pages 174–187. Springer-Verlag, 2005.

[13] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX '09: Proc. 18th Usenix Security Symposium*, 2009.

[14] Malheur: Automatic Analysis of Malware Behavior. `http://www.mlsec.org/malheur`.

[15] Microsoft. Portable Executable and Common Object File Format Specification, 2008. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

[16] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *S&P '05: Proc. 25th IEEE Symposium on Security and Privacy*, pages 226–241. IEEE Computer Society, 2005.

[17] Open Letter to RSA Customers. `http://www.rsa.com/node.aspx?id=3872`.

[18] M. Zubair Shafiq, S. Momina Tabish, F. Mirza, and M. Farooq. PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime. In *RAID '09: Proc. 12th International Symposium on Recent Advances in Intrusion Detection*, pages 121–141. Springer-Verlag, 2009.

[19] S. Sidiroglou and A.D. Keromytis. A Network Worm Vaccine Architecture. In *WETICE '03: Proc. 12th International Workshop on Enabling Technologies*, pages 220–225. IEEE Computer Society, 2003.

[20] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *CCS '09: Proc. 16th ACM conference on Computer and Communications Security*, pages 635–647. ACM Press, 2009.

[21] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[22] S. Momina Tabish, M. Zubair Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *CSI-KDD '09:*

Proc. ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics, pages 23–31. ACM Press, 2009.

[23] VirusTotal: Online Virus, Malware and URL Scanner. `http://www.virustotal.com`.

[24] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

Auto-learning of SMTP TCP Transport-Layer Features for Spam and Abusive Message Detection

Georgios Kakavelakis
Naval Postgraduate School

Robert Beverly
Naval Postgraduate School

Joel Young
Naval Postgraduate School

Abstract

Botnets are a significant source of abusive messaging (spam, phishing, etc) and other types of malicious traffic. A promising approach to help mitigate botnet-generated traffic is signal analysis of transport-layer (i.e. TCP/IP) characteristics, e.g. timing, packet reordering, congestion, and flow-control. Prior work [4] shows that machine learning analysis of such traffic features on an SMTP MTA can accurately differentiate between botnet and legitimate sources. We make two contributions toward the *real-world* deployment of such techniques: i) an architecture for real-time on-line operation; and ii) auto-learning of the unsupervised model across different environments without human labeling (i.e. training). We present a "SpamFlow" SpamAssassin plugin and the requisite auxiliary daemons to integrate transport-layer signal analysis with a popular open-source spam filter. Using our system, we detail results from a production deployment where our auto-learning technique achieves better than 95 percent accuracy, precision, and recall after reception of $\approx$ 1,000 emails.

1 Introduction

"Botnets" are distributed collections of compromised networked machines under common control [7]. Automated methods scan, infect, or socially engineer vulnerable hosts in order to incorporate them into the botnet. Botnets provide a formidable computing and communication platform by harnessing the power of thousands, or even millions, of nodes for a common collective purpose [21]. Unfortunately, that purpose is often malicious and economically or politically motivated.

As one common use scenario, botnets account for more than 85 percent of all abusive electronic mail (including spam, phishing, malware, etc) by one estimate [14]. Botnet-based spamming campaigns are large and long-lived [20], with more than 340,000 botnet hosts involved in nearly 8,000 campaigns in one study [27]. The Messaging Anti-Abuse Working Group (MAAWG) coalition of service providers reported that across 500M monitored mailboxes in one quarter of 2007, 75 percent of all messages (almost 400 billion) were spam [18]. A subsequent 2010 MAAWG study reports the situation has *worsened*: abusive messages accounted for 89 percent of all electronic mail in a representative sample across many providers.

Abusive message traffic abounds on the Internet. This deluge of unwanted traffic is more than a mere nuisance: a broad survey of large service providers finds that abusive messages account for the largest fraction of expended operational resources [1]. Despite extensive research and operational deployments, attackers and attacks have evolved at a rate faster than the Internet's ability to defend. There remains ample room for improvement of in-production botnet attribution and mitigation.

One promising approach for mitigating botnet-generated abusive messaging is statistical traffic analysis. Prior work [4] shows that by using transport-layer traffic features, e.g. TCP retransmits, out-of-order packets, delay, jitter, etc., one can reliably infer whether the source of an email SMTP [16] flow is legitimate or originating from a member of a botnet. Botnets must send large volumes of abusive messages to remain financially viable. Because bots are frequently attached via asymmetric (low upload bandwidth) residential connections, they necessarily congest their local uplink – an effect that is remotely detectable. Perhaps most importantly, transport-layer classifiers are content (e.g. the words of the message itself) and IP reputation (e.g. blocklist) agnostic, facilitating privacy-preserving deployment even within the network core. Deployed on individual Mail Transport Agents (MTAs), such techniques can permit early-rejection of messages before application delivery, significantly reducing system load.

Thus far, research in transport-layer classification has been offline, where experimental data is examined *a pos-*

teriori. In this paper, we present the system engineering efforts required to integrate TCP transport features into the classification decisions of the popular open-source SpamAssassin [17] spam filter. A crucial obstacle to realizing such techniques is the ability to adequately train and build a model of normal and abusive traffic across a variety of operational environments. Rather than requiring human labeling or overly general models to build ground-truth, we exploit the auto-learning functionality of SpamAssassin. Our primary contributions include:

1. *On-line* and *real-time* transport-layer classification of live email messages on a production MTA.
2. Auto-learning of transport features to automatically learn the unsupervised model across different operating environments without human training.

The remainder of this paper describes related work (§2). From this foundation, we describe our system architecture and testing methodology (§3). We present production deployment results in §4 and discuss their implications (§5). We conclude by outlining future work.

2 Related Work

Recent research efforts have shown great promise in understanding the character and behavior of botnets. While these proposed solutions are currently effective, they frequently rely on brittle heuristics and unreliable indicators. For instance, Xie et al. provide a system [27] to identify and characterize botnets using an automatic technique based on discerning spam URLs in email. Other research relies on IP addresses as indicators [29]. However, malicious botnet IP addresses are highly dynamic as new hosts are compromised, existing hosts receive new DHCP leases, or sources are spoofed [3]. Indeed, "fresh" IP addresses, i.e. those not in real-time blocklists, are a valuable commodity. Similarly, DNS is a poor identifier of malicious hosts given the prevalence of botnets employing DNS fast-flux [5] techniques to distribute load among redirectors, survive node failures, and obfuscate back-end hosting infrastructure.

A large body of work examines *network-layer* (IP) properties of botnets. Ramachandran et al. [22] characterize spamming behavior by correlating data collected from three sources: a sinkhole, a large e-mail provider, and the command and control of a Bobax botnet. By focusing on network-level properties including: i) IP address space from which spam originates; ii) the autonomous system (AS) that sent spam messages to their sinkhole; and iii) BGP route announcements, they show that spam and legitimate e-mail originate from the same portion of the IP address space. Thus, IP addresses are not a reliable indicator of malicious or abusive nodes.

Subsequent work from Hao et al. [11] demonstrates that AS alone as a feature may cause a large rate of false positives. They achieve better results by extracting lightweight features from network-level properties such as geodesic distance between sender and receiver, sender IP neighborhood density, probability ratio of spam to ham at the time of day the message arrives, the AS number of the sender, and the status of open ports on the sender machine. Further studies [15, 28] have shown that a spammer can evade such techniques by advertising routes from a forged AS number [11].

Schatzmann et al. [24] similarly focus on network-level characteristics of spammers, but from the perspective of an AS or service provider. Their idea is to passively collect the aggregate decisions of a large number of e-mail servers that perform some level of pre-filtering (e.g. blocklisting). Using passive flow collection to gather byte, packet, and packet size counts, this aggregated knowledge can enhance spam mitigation.

Commercial vendors expend considerable effort dividing the Internet IP address space into regions, with particular attention given to identifying residential broadband addresses. By discriminating against residential hosts, the hope is to block traffic from nodes that should not be sourcing email in the first place. This approach is both brittle and raises architectural misgivings in the form of arbitrarily discriminating against classes of users without prior provocation. Such residential blocking may have implications on notions of network neutrality as neutrality legislation catches up with technology.

In contrast to these spam detection and mitigation techniques, Beverly and Sollins [4] present a content and IP reputation agnostic scheme based on statistical signal analysis of the *transport* (TCP) traffic stream. The premise is that spammers must send large volumes of e-mail to be effective, causing constituent network links to experience contention and congestion. Such congestion effects are particularly prominent for many botnet hosts which reside on residential broadband connections where there are large gateway buffers [12] and asymmetric bandwidth. Transport-layer properties such as the number of lost segments and round trip time (RTT) therefore exhibit different distributions, permitting discrimination between spam and legitimate behavior. Among many TCP features, their analysis found that RTT and minimum-congestion window are the most discriminatory. This transport-only classifier exhibits more than 90 percent accuracy and precision on their data.

Follow-on work to [4] explore similar ideas, including the use of lightweight single-TCP/SYN passive operating system signatures at the router-level [10]. Ouyang et al. [19] conduct a large-scale empirical analysis of transport-layer characteristics on over 600,000 messages. Among tested features, their analysis similarly finds the

three-way-handshake latency, time-to-live (TTL), and inter-packet idle time and variance most discriminating for ham versus spam. These features remain stable over time, yielding 85-92 percent classification accuracy.

Based on the encouraging results of this body of prior work, we endeavor to take a step toward the *real-world deployment* of transport-classifier based botnet detection and abusive traffic mitigation techniques.

3 System Architecture

The TCP/IP network stack logically divides functionality between layers. As a result, applications do not normally have access to lower-layer features. For example, TCP (implemented in the kernel or lower) provides an abstraction of a reliable and in-order data stream to the application via a socket interface. Applications are removed from the details of packet arrival timing, ordering, TTL, etc. Thus, our design must collect, on a per-message basis, transport-layer traffic characteristics and expose them up the stack to the SpamFlow (SF) plugin. This section describes our system architecture and the interaction between various components: SpamAssassin, SpamFlow, and the SpamFlow plugin.

3.1 Overview

We start with an overview of our SpamFlow system architecture, shown in Figure 1. For clarity of exposition, we describe all functionality as being co-located with the Mail Transport Agent (MTA); however, the components can easily be distributed across different machines. The system is comprised of four main components: SpamAssassin, the SpamFlow traffic feature extraction engine, the SpamFlow plugin, and the classification software – referred to as SpamAssassin, SpamFlow, SF plugin, and classifier respectively.

Every message received by the MTA is processed by SpamAssassin and then piped to the plugin. Simultaneously, SpamFlow continuously and promiscuously listens on the network interface, capturing SMTP packets via the pcap API [13], aggregating packets into flows, and computing the relevant traffic statistics (e.g. TCP retransmits, out-of-order packets, delay, jitter, etc.). The plugin queries SpamFlow with the message's identifier in order to retrieve the flow-level transport features corresponding to that message. Next, the plugin sends the message's transport feature vector to the classifier. In response, the classifier returns a binary or probabilistic prediction (depending on the classifier employed) that then influences the final score of the message, and hence the final disposition. We describe each component in more detail in the following subsections.

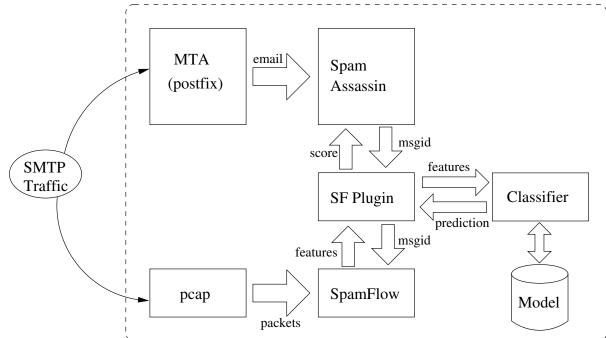


Figure 1: SpamFlow system architecture: transport-layer features are aggregated on a per-flow basis. The SpamFlow SpamAssassin plugin uses XML-RPC to obtain each message's feature vector which is then sent to the classifier. Predictions are relayed to the plugin and integrated into the final SpamAssassin message score.

3.2 SpamAssassin

SpamAssassin [17] is an open-source, rule and content learning-based spam filter. Each rule is assigned a weight by a perceptron algorithm [25] and then the weighted scores are summed to produce an overall score for each message. The classification process involves comparing the overall score with a user-defined threshold t (which defaults to a value that maximized performance on a broadly representative training sample). If the score is above t, then the message is classified as spam; otherwise, as legitimate. SpamAssassin is modular and extensible for adding other filtering techniques. Popular plugins include real-time block lists (RBLs), domain-keys, permit lists, collaborative filtering, learning-based techniques (e.g. naïve Bayes), and others.

Furthermore, SpamAssassin features a threshold-based mode in which new exemplar emails trigger an automatic retraining process. While SpamAssassin refers to this retraining as "auto-learning," this is typically known as "online" or "iterative" learning in machine learning. The primary difference is that advanced iterative learning approaches modify the classification model to account for new emails, whereas in auto-learning the entire model is rebuilt each time. In SpamAssassin auto-learning, a previously unseen message is used to retrain the model if it receives a score greater than τ^+ (assumed spam) or less than τ^- (assumed non-spam). For example, when a message exceeds these threshold values, SpamAssassin rebuilds the model of the built-in naïve Bayes classifier, and classifies subsequent messages with the newly updated model.

```
--- src/smtpd/smtpd.c.orig
+++ src/smtpd/smtpd.c
@@ -2807,9 +2807,9 @@
 */
 if (!proxy || state->xforward.flags == 0) {
   out_fprintf(out_stream, REC_TYPE_NORM,
-    "Received: from %s (%s [%s])",
+    "Received: from %s (%s [%s:%s])",
     state->helo_name ? state->helo_name : state->name,
-    state->name, state->rfc_addr);
+    state->name, state->rfc_addr, state->port);
```

Figure 2: Postfix modification to support traffic identifiers

```
--- received.c.orig
+++ received.c
@@ -44,2 +44,3 @@
 char *remoteip;
+char *remoteport;
 char *remotehost;
@@ -63,2 +64,5 @@
   safeput(qqt,remoteip);
+  remoteport = getenv("TCPREMOTEPORT");
+  qmail_puts(qqt,":");
+  safeput(qqt,remoteport);
   qmail_puts(qqt,")\n  by ");
```

Figure 3: qmail modification to support traffic identifiers

3.3 SpamFlow

SpamFlow [4] is our network analyzer. Using libpcap [13], SpamFlow promiscuously listens on the network interface and builds source host/port flows (the destination MTA address is constant and known and thus not part of the flow tuple). As SMTP flows complete, either via an explicit TCP termination handshake or via timeout, SpamFlow extracts transport-layer features for each as detailed extensively in [4]. SpamFlow listens for XML queries for a particular flow's IP and port, responding in kind with the features for that flow.

We explored two options for uniquely identifying messages to correlate between messages and their constituent flow data. First, every message contains a unique message string ("`Message-ID`" in the header) [23] to facilitate replies, threading, etc. Using deep packet inspection, SpamFlow could reassemble email messages from the packet payloads to uniquely identify each flow by `Message-ID`. The immediate downside to using the message identification field is that doing so removes the benefit of only examining packet header statistics: namely privacy and efficiency.

Instead, we opt to follow a simpler approach and use remote host IP address and ephemeral port number as the message identifier. These fields are readily available without any transport reassembly and are, in general, unique. Naturally, IP address and port tuples are reused (there is a maximum of only 2^{16} unique TCP client-side ephemeral ports). For a tuple collision to occur in SpamFlow, two identical flows must arrive within less time than the messages can be delivered to the MTA and processed by SpamAssassin, i.e. on the order of a few seconds. Not only is this in violation of the TCP time wait procedure, we do not observe any duplicate flows within such short time periods in our empirical data.

The final detail is how to expose the message identifier to the plugin so it can query SpamFlow. We modify our MTA server to add the $(IP_address, TCP_port)$ identification tuple of the remote MTA to the header of each incoming e-mail. The actual MTA code modifications are small and straightforward. For reference we provide the code changes for the popular Postfix and qmail MTAs in Figures 2 and 3.

3.4 SpamFlow Plugin

SpamFlow does not operate as a standalone MTA or spam classifier. Therefore, we integrate it with an existing one. We select SpamAssassin [17] because it is open source and widely used; for instance, the commercial Barracuda [2] network appliance is based on SpamAssassin. Importantly, SpamAssassin employs a modular architecture that allows developers to extend its functionality through plugins. As SpamAssassin is written in Perl, we develop a small, lightweight SpamAssassin Perl plugin tying the various components of Figure 1 together. In real-time, as e-mail messages are routed through the SpamFlow plugin, it scores them using a previously learned model of transport features. This score, in combination with the scores from other rules, provides a final message disposition.

The plugin acts as the controller of the system and binds the traffic analysis engine and the classifier together. First, the plugin provides SpamFlow with the 2-tuple identifier of the message under inspection and receives in return the corresponding message's transport-layer features. After obtaining the features, the plugin passes them to a logically distinct machine learning classifier and retrieves the corresponding prediction. Figure 4 shows an example where the MTA added the message identifier (here, `77.239.18.226:37689`) and the plugin attached SpamFlow's transport feature vector to the message's headers.

Between components, we use XML-RPC [26] to communicate. XML-RPC is a simple protocol that allows communication between procedures running in different applications or machines. Specifically, the client uses the HTTP-POST request to pass data to the server; the server in return sends an HTTP response. In our implementation, we register the classifier with a `classify` procedure that takes as input the features. Thus, the plugin

```
From Josephine@rsi.com Tue Feb 01 23:21:58 2011
Return-Path: <Josephine@rsi.com>
X-Spam-Checker-Version: SpamAssassin 3.3.1 (2010-03-16) on ralph.rbeverly.net
X-Spam-Level: ******
X-Spam-Status: Yes, score=6.9 required=5.0 tests=BAYES_50,RCVD_IN_XBL,HTML_MESSAGE,
          SPAMFLOW, UNPARSEABLE_RELAY autolearn=no version=3.3.1
X-Spam-Spamflow-Tag: 3792891725:37689,12,10,0,0,0,0,1,1,0,53248,34.464852,0.162818,
          120.441156,148.297699,51.891697,5840,48,1,64
Received: (qmail 30920 invoked from network); 1 Feb 2011 23:21:57 -0000
Received: from cm-static-18-226.telekabel.ba (77.239.18.226:37689)
Received: from vdhvjcvivjvbwyhscvfwq (192.168.1.185) by bluebellgroup.com (77.239.18.226) with Microsoft SMTP
Message-ID: <4D489025.504060@etisbew.com>
Date: Wed, 2 Feb 2011 00:20:48 +0100
From: Essie <Essie@hermes.com>
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.12)
```

Figure 4: Example email message headers with transport features added by the SpamFlow system

sends the HTTP-POST request with the name of the classify procedure along with the features, as comma separated values forming a string[1], and receives via HTTP response the classification prediction from the classifier.

Not only is XML-RPC simple and standardized, it allows the classifier to potentially operate on a different machine from SpamFlow, which in the future could allow the XML-RPC classifier to serve many SpamFlow instances in a multi-threaded fashion and distribute load. Further, all popular programming languages provide XML-RPC APIs, notably allowing us to use our language of choice for the various tasks. In our specific implementation, we develop SpamFlow in C++ while the classifier is a Python daemon.

3.5 Classification Engine

The final component of the system architecture is the traffic classification engine which we implement using the open source Orange [9] machine learning and data mining Python package. While the details of the machine learning algorithms are out of scope for this paper, we note that Orange includes a variety of algorithms and statistical modules for performance evaluation.

Our classifier implementation experiments with three machine-learning algorithms: naïve Bayes, decision trees (specifically, the C4.5 algorithm), and support vector machines (SVM). These three algorithms are broadly representative of different classes of learning strategies and allow us to evaluate both system classification performance, generality, and system speed.

4 Results

This section first describes results from load testing the SpamFlow system in a controlled laboratory environment in order to understand its practical feasibility. We then detail performance results using auto-learning of transport features in a live production environment.

4.1 Load Testing

To understand the system-level performance of our SpamFlow design as outlined in §3, we create the controlled testing environment depicted in Figure 5. One host runs the SpamFlow system and is physically connected to a second traffic sourcing host. The traffic sourcing host implements our custom e-mail "replayer" application and a modified Dummynet [6] network emulator.

The replayer reads from the TREC public email corpus [8] of 92,187 messages, of which 52,788 are spam and 39,399 are legitimate. For each message, the replayer: 1) extracts the headers and adds as recipient a valid user of our virtual-network domain; 2) establishes an SMTP session with the MTA (Postfix) of the SpamFlow system under test; 3) sets the differentiated services code point (DSCP) in the IP header of each message according to the ground truth label (spam or ham); 4) uses the standard SMTP protocol to transmit the message.

We set the DSCP differently for spam and non-spam messages in order to influence the emulated network behavior. Our goal is to coarsely simulate the characteristics that botnet-generated spam traffic exhibits, such as TCP timeouts, retransmissions, resets, and highly variable RTT estimates. For our evaluation, we select Dummynet [6], a publicly-available tool that enables introduction of delay, loss, bandwidth and queuing constraints, etc. for packets passing through virtual network links. In our testing setup, Dummynet applies different queuing, scheduling, bandwidth, delay, loss, etc. depending on the DSCP bits which correspond to email type. Dummynet emulates a only fixed propagation delay. We therefore modify it to generate random delays drawn from a normal distribution with a mean delay of $\mu = 150$ms with $\sigma = 50$ms standard deviation for spam

[1]The CSV string is used for expediency; in the future, we plan to use individual XML identifiers for each feature.

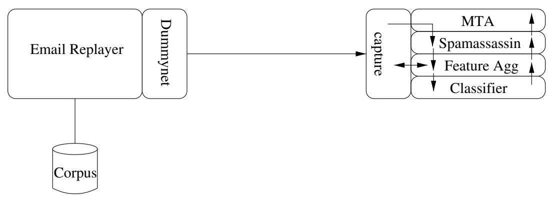


Figure 5: Laboratory testing environment: enabling tightly controlled and easily configurable *repeatable* experiments. The replayer application replays an email corpus while Dummynet emulates different network behavior to mimic botnet and legitimate traffic. Using the testbed, we load test and debug SpamFlow.

traffic that originates from the replayer, and a $\mu = 40$ms delay with $\sigma = 25$ms for legitimate traffic in both directions. We introduce delay in legitimate traffic in order to avoid overfitting our learned statistical model. These delays need not be precise as they are intended to merely mimic a congested environment. To emulate timeouts, retransmissions, and resets, we apply a random-packet-drop policy on the Dummynet pipe.

Note that we disable all SpamAssassin rules requiring network access, e.g. real-time blocklists, as such rules are dynamic and thus sensitive to dates and time.

While we recognize that our modifications to Dummynet only partially emulate a congested network (for example, loss events are independent – an assumption that does not hold true in a real queue), our goal in the emulation environment is to enable reproducible testing. Thus, we use the environment to emulate high-rate traffic and evaluate performance, throughput, system load, etc. on representative traffic. Section 4.3 goes on to detail real-world performance on live production traffic.

Table 1 shows the performance of the three classifiers with respect to training time. C4.5 has the smallest training time. SVM, on the other hand, has the largest training time, due to the more complex decision model.

We then examine throughput: the rate at which the system is able to classify and process emails from the replayer. Naïve Bayes, C4.5, and SVM achieve 1,300, 1,000, and 700 messages per second throughput respectively in our environment. Naïve Bayes provides the highest throughput, likely due to its simple decision rule.

Many factors impact throughput; our intent is to understand the *relative* performance of each classifier and to establish real-world feasibility. The takeaway from these measurements is that, taking into account the relative independence of our system from the classification method, we can select the classification model that fits our needs. For example, the low training time of C4.5 makes it a good candidate when we need to retrain often.

Table 1: SpamFlow training time (sec) as a function of classifier type and sample size

	Training Samples			
Classifier	**10**	**100**	**1000**	**10,000**
Naive Bayes	0.88	15.02	105.45	104.84
C4.5	0.15	0.96	16.02	29.80
SVM	0.72	12.69	224.25	260.02

4.2 Production Environment

Live testing is important because it reveals how the system interacts with possibly unknown features of the external environment. We deployed our system in a live environment at our university for a small domain from January 25, 2011 to March 2, 2011 and collected a trace of 5,926 e-mail messages.

Ground truth was first established using an unmodified SpamAssassin version 3.3.1 instance without transport-layer traffic features, i.e. with only the default built-in rules and content analysis. We then manually examined all the legitimate ham messages and relabeled those that were false negatives. We manually sampled the spam messages to eliminate false positives and establish reasonable ground truth. While the volume of traffic captured is small, our intent in this experiment is to establish the ability to auto-learn the transport-layer features in a production environment and ascertain the resulting classification performance. We envision larger-scale, higher-volume live testing in the future.

Auto-learning is the incremental process of building the classification model based on exemplar e-mail messages whose scores exceed certain threshold values. In our case, we use features of e-mail messages otherwise classified via orthogonal methods as having very high or very low scores (for instance, those emails whose content triggers many of SpamAssassin's rule-based indicators). Specifically, we explicitly retrain the classifier's model each time a new message obtains an especially high or low score from the other SpamAssassin methods (rule- and Bayesian-word based); i.e. a score above or below set thresholds. After retraining is complete, we evaluate performance iteratively on subsequent messages until a new message arrives with a score above or below the threshold, triggering retraining again.

Our thresholds selection is based on empirical spam and ham SpamAssassin score distributions. Spam message scores follow a normal distribution with $\mu = 16.3$ and $\sigma = 7.7$, whereas scores of legitimate messages have a mean of $\mu = 1.3$, but are skewed left. Therefore, for the legitimate messages we first experiment with a threshold $\tau^+ = 16$ and $\tau^- = 1$, which allows the classifiers to be trained on an approximately even fraction of training and test examples: a total of 2,683/5,590 (48.0%) spam and

296/436 (67.9%) ham messages.

We canonically call spam a "positive" and ham a "negative" to indicate disposition. Correct predictions result in either a true positive (tp) or true negative (tn). A spam message that is mispredicted as ham produces a false negative (fn), while a ham message misclassified as spam produces a false positive (fp). Note that false positives in email filtering are particularly expensive for users as there is a high cost to missing or discarding legitimate messages. As performance metrics, we consider accuracy, precision, recall, specificity, and F-score:

$$accuracy = \frac{tp+tn}{tp+fp+tn+fn} \quad (1)$$

$$precision = \frac{tp}{tp+fp} \quad (2)$$

$$recall = \frac{tp}{tp+fn} \quad (3)$$

$$specificity = \frac{tn}{fp+tn} \quad (4)$$

$$F-score = 2\left(\frac{precision*recall}{precision+recall}\right) \quad (5)$$

All of these metrics are important to consider to properly understand system performance. For instance, accuracy is misleading if the underlying class prior is heavily skewed: if 95% of the messages are in fact spam, then a deterministic classifier that always predicts "spam" will achieve seemingly high 95% accuracy without any learning. Precision therefore measures, among messages predicted to be spam, the fraction that are truly spam. Recall measures the influence of misclassified spam messages, i.e. is a metric of the classifier's ability to detect spam. Specificity, or true negative rate, determines how well the classifier is differentiating between false positives and true negatives. Finally, because there is a natural tension between achieving high precision and high recall, a common metric is F-Score which is simply the harmonic mean of precision and recall.

4.3 Production Testing

Figure 6 shows the classification performance metrics of the three classifiers we implement in SpamFlow as a function of cumulative training samples received. Figure 6 therefore depicts the classifiers' auto-learning over time as new exemplar training messages are received.

Figure 6(a) displays cumulative accuracy for each classifier over time and includes the spam prior. The spam prior is simply the fraction of all training emails that are spam. A naïve classifier could simply predict the prior, so values above the prior indicate true learning. We observe both decision trees and SVMs providing greater than 95 percent accuracy. Figure 6(c) similarly shows decision tree and SVM providing high F-scores, indicative of very good performance using only transport-layer features. Of note is that this level of performance is achieved after receiving only 100-200 messages. The weakness in SpamFlow only using traffic characteristics appears in the specificity, Figure 6(e), where false positives drive our best specificity down to approximately 75 percent.

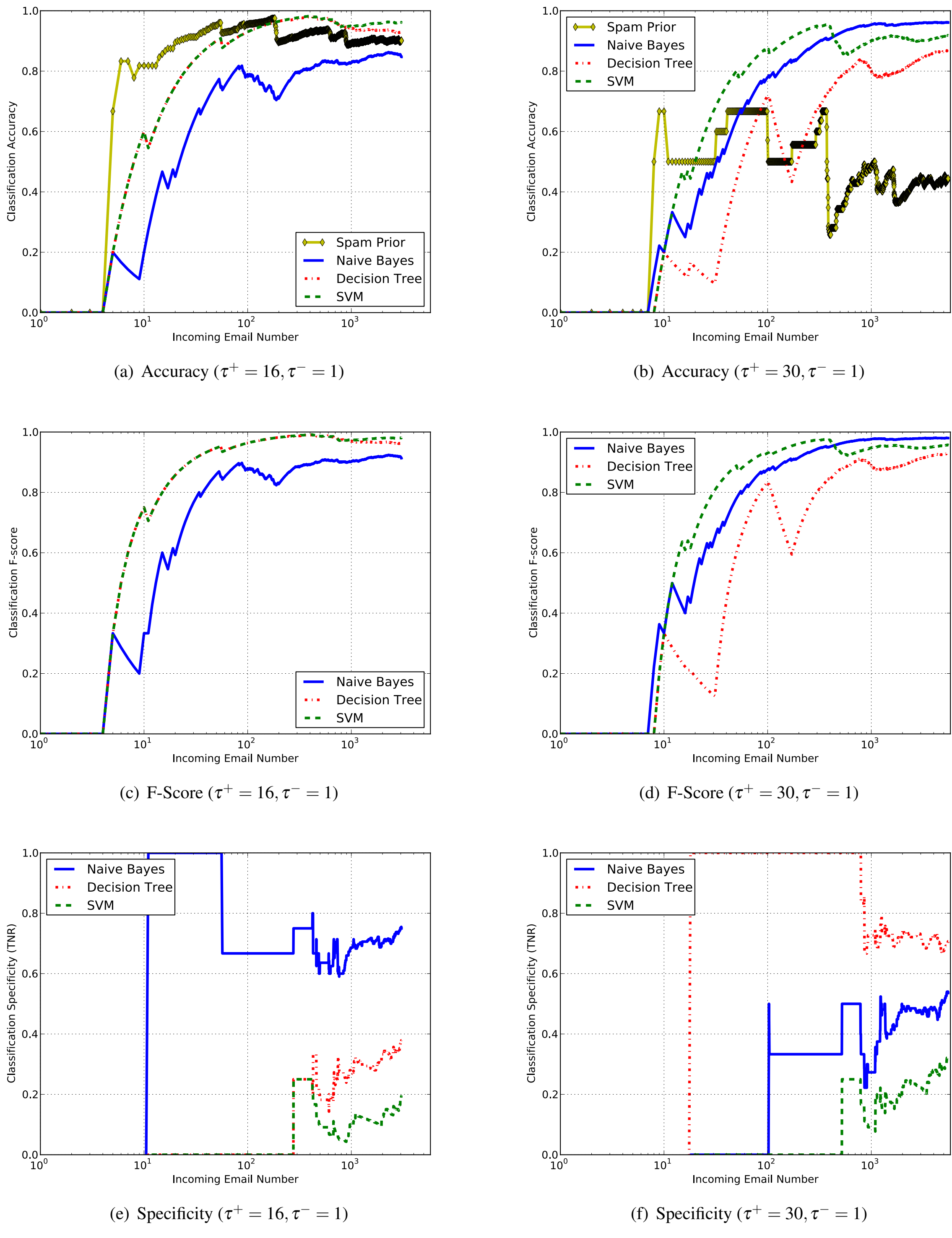
To better understand the sensitivity of our auto-learning results to the imposed thresholds τ, we experiment with a spam threshold two deviations above the mean: $\tau^+ = 30$. By increasing the spam threshold, the SpamFlow auto-learning uses fewer spam-training examples. However, we expect to have higher confidence in their true disposition of spam with the higher threshold. Important to our evaluation, $\tau^+ = 30$ has the effect of balancing the training complexion so that there is not a strong class prior: 227 exemplar spam messages and 296 exemplar ham messages.

With the spam score threshold raised to $\tau^+ = 30$, Figure 6(b) shows that the spam prior is now close to 50 percent, removing any training class bias. SVM and naïve Bayes still achieve greater than 90 percent accuracy. Again, clearly the auto-learning behavior is working with performance steadily increasing over time and greatly outperforming the spam prior. As with the lower threshold, Figure 6(d) demonstrates very high F-Scores for all of the classifiers.

Figure 6(f) highlights the challenge in false positives. However, the most specific classifier, the decision tree algorithm, is also highly accurate and precise. With machine learning there is an inherent trade off between achieving very high true positive rates and keeping false positive rates low. Our results demonstrate the best compromise with the higher auto-learning threshold and the use of decision trees.

Finally, we perform an initial investigation into whether the *combined* votes of SpamAssassin and SpamFlow lead to overall improved performance. We experiment with adding 0.2 (experiment 1) and with adding 1.0 (experiment 2) to the final score if SpamFlow predicts a spam message on the basis of transport traffic characteristics. Otherwise, we subtract 1.0 from the final score. This crude weighting does not leverage SpamFlow's confidence in the prediction, and does not properly weight the vote in accordance with SpamAssassin's other rules. We leave complete integration of SpamFlow's predictions with SpamAssassin's voting as future work.

Table 2 shows the confusion data for SpamAssassin alone, SpamFlow alone, and the combination. In the first combined vote, we achieve better performance with the same number of false positives. In the second combined vote, we achieve even better performance, but at the cost of false positives. In all cases, the combination increases the overall F-score.



(a) Accuracy ($\tau^+ = 16, \tau^- = 1$)

(b) Accuracy ($\tau^+ = 30, \tau^- = 1$)

(c) F-Score ($\tau^+ = 16, \tau^- = 1$)

(d) F-Score ($\tau^+ = 30, \tau^- = 1$)

(e) Specificity ($\tau^+ = 16, \tau^- = 1$)

(f) Specificity ($\tau^+ = 30, \tau^- = 1$)

Figure 6: Auto-learning classification results for three SpamFlow classifiers on live production traffic as a function of cumulative exemplar training messages received.

Table 2: Confusion data comparing SpamAssassin performance with and without SpamFlow auto-learning

	tp	fp	tn	fn	F-Score
SpamAssassin	5288	3	137	87	0.991
SpamFlow	5224	65	75	151	0.980
SA+SpamFlow(1)	5299	3	137	76	0.992
SA+SpamFlow(2)	5335	19	121	40	0.995

5 Discussion

Can spammers adapt and avoid a transport-based classification scheme? By utilizing one of the fundamental weaknesses of spammers, their need to send large volumes of spam on bandwidth constrained links, we believe SpamFlow is difficult for spammers to evade. A spammer might send spam at a lower rate or upgrade their infrastructure in order to remove congestion effects from their flows. However, either strategy is likely to impose monetary and time costs on the spammer.

Of note is that our techniques work equally well in IPv6 as the TCP transport-layer characteristics SpamFlow relies on in IPv4 are the same in IPv6. The fact that SpamFlow is IP address agnostic suggests that it may be an even more important technique in an IPv6 world where the large address space is difficult to reliably map.

One possible limitation of SpamFlow is that it may be unable to distinguish between a botnet host sending large volumes of spam and traffic from a host that is simply busy, or on a congested subnetwork. However, other transport-layer features are decoupled from congestion, for instance a CPU-bound bot host will perform TCP flow control and advertise a small receiver window – an effect that SpamFlow uses as part of its decision process.

Further, SpamFlow detects hosts that send volumes of email that exceed the local uplink and processing capacity. Personal, home or small business servers do not have the same volume requirement as spammers and thus are unlikely to induce the same TCP congestion effects we observe. In reality, there is a value judgment that makes SpamFlow practical and reasonable. Specifically, users who wish to ensure that their emails are delivered typically invest in suitable infrastructure, contract with an outside provider or use their service provider's email systems. Companies are not sourcing large amounts of crucial email from hosts attached by consumer-grade connections. The vast majority of home users utilize their provider's email infrastructure or employ popular web-based services. Thus, SpamFlow only discriminates against sources that are both poorly connected *and* injecting large volumes of mail.

However, in future work, we plan to experiment with the sensitivity of SpamFlow to false positive originating from congestion induced by other nodes and other applications. We believe there will remain adequate discriminatory signal to discern botnet hosts. Even when SpamFlow does mispredict, our results show that combining SpamFlow with other classifiers leads to improved performance and can overcome instances of false positives by individual classifiers.

6 Conclusions and Future Work

This research implemented the necessary infrastructure to perform real-time, on-line transport-layer classification of email messages. We plan to distribute our system as part of the third-party SpamAssassin plugin library in order to facilitate widespread deployment, impart impact on abusive messaging traffic, and to refine the system.

We detail the system architecture to integrate network transport features with SpamAssassin, an MTA, and a classification engine. Our testing reveals that the system can handle realistic traffic loads. Of note, we tackle the bootstrapping problem of obtaining representative network traffic on a per-network basis by leveraging auto-learning to automatically train on exemplar messages.

Using our techniques, we achieve accuracy, precision, and recall performance greater than 95 percent after receiving only $\approx 2^{10}$ messages during live, real-world production testing. We emphasize that these results come from observing *only* network traffic features; in actual deployment, the SpamFlow plugin will, as with other parts of the SpamAssassin system, place a weighted vote. Overall performance will likely improve using traditional features in addition to network traffic features.

We note, however, that our live-testing corpus is small. Our intent in this work was to demonstrate the practical feasibility of using transport network traffic features. In future work, we plan to investigate SpamFlow's performance and scalability in large, production systems against much larger volumes of traffic. Our hope is to enable the practical deployment of transport-layer based abusive traffic detection and mitigation techniques to system administrators.

Finally, we observe that the distributed computing platform offered by botnets enables a wide variety of attacks and scams beyond abusive email. Beyond messaging abuse, botnets are employed in phishing attacks, scam infrastructure hosting, distributed denial-of-service (DDoS) attacks, and more. For example, some botnets effectively provide a Content Distribution Network (CDN) for hosting scam infrastructure. Botnet CDNs are used to host web sites (e.g. landing sites for ordering prescription pharmaceuticals or redirection servers), distribute malicious code, and a variety of other nefarious purposes. Still other botnets are employed to perform dictionary attacks against servers, brute force or other-

wise solve CAPTCHAs [30], etc. in order to create accounts on social network sites and further spread abusive traffic via multiple distribution channels.

We believe transport-layer techniques generalize to any botnet generated traffic, including phishing attacks, scam infrastructure hosting, DDoS, dictionary attacks, CAPTCHA solvers, etc. In future research, we wish to investigate using transport-level traffic analysis to identify a variety of botnet attacks and bots themselves.

Acknowledgments

The authors would like to thank Geoffrey Xie, Le Nolan, Ryan Craven, and the anonymous reviewers. Special thanks to our shepherd Avleen Vig for invaluable feedback. This research was partially supported by a Cisco University Research Grant and by the NSF under grant OCI-1127506. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. government.

References

[1] ARBOR NETWORKS. Worldwide infrastructure security report, 2010. `http://www.arbornetworks.com/report`.

[2] BARACUDA NETWORKS. Baracuda spam and virus firewall, 2011. `http://www.barracudanetworks.com/`.

[3] BEVERLY, R., BERGER, A., HYUN, Y., AND CLAFFY, K. Understanding the efficacy of deployed internet source address validation filtering. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (2009), IMC '09.

[4] BEVERLY, R., AND SOLLINS, K. Exploiting transport-level characteristics of spam. In *Proceedings of the Fifth Conference on Email and Anti-Spam (CEAS)* (Aug. 2008).

[5] CAGLAYAN, A., TOOTHAKER, M., DRAPAEAU, D., BURKE, D., AND EATON, G. Behavioral analysis of fast flux service networks. In *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research* (2009).

[6] CARBONE, M., AND RIZZO, L. Dummynet revisited. *SIGCOMM Comput. Commun. Rev. 40* (April 2010), 12–20.

[7] COOKE, E., JAHANIAN, F., AND MCPHERSON, D. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet (SRUTI) Workshop* (July 2005).

[8] CORMACK, G., AND LYNAM, T. TREC public email corpus, 2007. `http://trec.nist.gov/data/spam.html`.

[9] DEMSAR, J., ZUPAN, B., LEBAN, G., AND CURK, T. Orange: From experimental machine learning to interactive data mining. In *Principles of Data Mining and Knowledge Discovery* (2004).

[10] ESQUIVEL, H., MORI, T., AND AKELLA, A. Router-level spam filtering using tcp fingerprints: Architecture and measurement-based evaluation. In *Proceedings of the Sixth Conference on Email and Anti-Spam (CEAS)* (2009).

[11] HAO, S., SYED, N. A., FEAMSTER, N., GRAY, A. G., AND KRASSER, S. Detecting spammers with snare: spatio-temporal network-level automatic reputation engine. In *Proceedings of the 18th conference on USENIX security symposium* (2009).

[12] HÄTÖNEN, S., NYRHINEN, A., EGGERT, L., STROWES, S., SAROLAHTI, P., AND KOJO, M. An experimental study of home gateway characteristics. In *Proceedings of the 10th annual conference on Internet measurement*, pp. 260–266.

[13] JACOBSON, V., LERES, C., AND MCCANNE, S. Tcpdump, 1989. `ftp://ftp.ee.lbl.gov`.

[14] JOHN, J. P., MOSHCHUK, A., GRIBBLE, S. D., AND KRISHNAMURTHY, A. Studying spamming botnets using botlab. In *Proceedings of USENIX NSDI* (Apr. 2009).

[15] KARLIN, J., FORREST, S., AND REXFORD, J. Autonomous security for autonomous systems. *Computer Networks 52*, 15 (2008). Complex Computer and Communication Networks.

[16] KLENSIN, J. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), Oct. 2008.

[17] MASON, J. Filtering spam with spamassassin. In *Proceedings of SAGE-IE* (Oct. 2002).

[18] MESSAGING ANTI-ABUSE WORKING GROUP. Email metrics report, 2011. `http://www.maawg.org/about/EMR`.

[19] OUYANG, T., RAY, S., RABINOVICH, M., AND ALLMAN, M. Can network characteristics detect spam effectively in a stand-alone enterprise? In *Passive and Active Measurement* (2011).

[20] PATHAK, A., QIAN, F., HU, Y. C., MAO, Z. M., AND RANJAN, S. Botnet spam campaigns can be long lasting: evidence, implications, and analysis. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems* (2009), ACM, pp. 13–24.

[21] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets* (2007), USENIX Association, pp. 5–5.

[22] RAMACHANDRAN, A., AND FEAMSTER, N. Understanding the network-level behavior of spammers. In *Proceedings of ACM SIGCOMM* (Sept. 2006).

[23] RESNICK, P. Internet Message Format. RFC 2822 (Proposed Standard), Apr. 2001.

[24] SCHATZMANN, D., BURKHART, M., AND SPYROPOULOS, T. Inferring spammers in the network core. In *Proceedings of the 10th International Conference on Passive and Active Network Measurement* (2009), pp. 229–238.

[25] STERN, H. Fast spamassassin score learning tool, Jan. 2004. `http://svn.apache.org/repos/asf/spamassassin/trunk/masses/README.perceptron`.

[26] WINER, D. XML-RPC specification, Apr. 1998. `http://www.xmlrpc.com/spec`.

[27] XIE, Y., YU, F., ACHAN, K., PANIGRAHY, R., HULTEN, G., AND OSIPKOV, I. Spamming botnets: signatures and characteristics. *SIGCOMM Comput. Commun. Rev. 38*, 4 (2008), 171–182.

[28] ZHAO, X., PEI, D., WANG, L., MASSEY, D., MANKIN, A., WU, S. F., AND ZHANG, L. An analysis of bgp multiple origin as (moas) conflicts. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement* (2001), pp. 31–35.

[29] ZHAO, Y., XIE, Y., YU, F., KE, Q., YU, Y., CHEN, Y., AND GILLUM, E. Botgraph: large scale spamming botnet detection. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (2009), pp. 321–334.

[30] ZHU, B. B., YAN, J., LI, Q., YANG, C., LIU, J., XU, N., YI, M., AND CAI, K. Attacks and design of image recognition captchas. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), CCS '10.

Using Active Intrusion Detection to Recover Network Trust

John F. Williamson, *Dartmouth College*
Sergey Bratus, *Dartmouth College*
Michael E. Locasto, *University of Calgary*
Sean W. Smith, *Dartmouth College*

Abstract

Most existing intrusion detection systems take a passive approach to observing attacks or noticing exploits. We suggest that *active* intrusion detection (AID) techniques provide value, particularly in scenarios where an administrator attempts to recover a network infrastructure from a compromise. In such cases, an attacker may have corrupted fundamental services (e.g., ARP, DHCP, DNS, NTP), and existing IDS or auditing tools may lack the precision or pervasive deployment to observe symptoms of this corruption. We prototype a specific instance of the active intrusion detection approach: how we can use an AID mechanism based on packet injection to help detect rogue services.

Tags: security, active intrusion detection, networking, trust relationships, recovery

1 Introduction

Existing network intrusion detection systems (e.g., Bro [35, 12], Snort [31]) typically take a passive approach to detecting attacks: they scan network packets and flows to match their content against known-malicious byte patterns (i.e., signatures). Such sensors are typically situated at the network edge or other traffic choke point rather than on individual hosts, and they rarely interpose on (i.e., inject packets or frames into) the actual connection or flow.

IDS systems rarely take an *active* approach to detecting malicious behavior or indicators within the network. By *active*, we mean that the sensor purposefully injects packets and data meant to perturb the state of the network, in essence becoming part of the various connections occurring on the network. Some existing IDS sensors may be "active" in the sense that they periodically scan some hosts or listen to some specific connections, or that they attempt to proactively firewall or quarantine hosts suspected of being malicious (for example, Network Access Control or NAC). To the best of our knowledge, most existing IDSs do not actively participate in network conversations to deduce end host behavior.

This hesitance may be due to the perceived danger of actively issuing network traffic designed to remotely diagnose the existence of malware or corrupted service on an end host or server (such traffic might have an adverse effect on benign hosts or servers).

In this paper, we suggest that the paradigm of active intrusion detection (AID) is relatively under-explored, and we offer an example of how such proactive scanning for malicious behavior at the network level can benefit a system administrator focused on recovering a network infrastructure from an attack that attempts to replace or spoof critical network services.

1.1 Motivation: Intrusion Recovery

Recovering a network infrastructure from an attack — particularly an attack that has compromised a large portion of the infrastructure [19] — is a complex, difficult, and time-consuming task. Furthermore, the administrator may not have much confidence in the services that remain running after the discovery of such a compromise. Because auditing and forensics are expensive processes (in terms of time and density of instrumentation), and such activities can be greatly curtailed because of the need to get the network back up and running, system administrators may have little information about what parts of the system remain trustworthy.

1.2 The Challenge of Recovering Trust

We see the fundamental difficulty in such a situation as the task of *recovering trust* in the network infrastructure. For example, if the DNS server has been compromised, users cannot trust that their DNS queries have not been tampered with. Similar trust relationships exist with ARP and DHCP along with other critical network ser-

vices. In essence, each protocol implies that the client trusts the server to relay correct information about the network properties. Likewise, the server trusts clients to act only on their own behalf.

Trust exists in many forms within the network. When a host accepts an offer of a DHCP address, it implicitly trusts the DHCP server it received the offer from. Similarly, when a switch participates in a bridge election, it implicitly trusts every other switch that is participating. This arrangement exists out of necessity, but can cause problems when the wrong entities are trusted.

Modern attacks have increased in sophistication; many now involve hijacking benign hosts and their network stacks for malicious use (which requires altering the normal behavior of the affected devices). Elements of the network infrastructure, such as routers and switches, are also attractive targets since network hosts frequently trust them implicitly. Compromising such machines can give an adversary a great deal of power without requiring him or her to attack very many machines. Such attacks are a useful way for an attacker to retain some level of control and spread, and one recent example[1] attempts to run a rogue DHCP service.

Without the ability to meaningfully trust the information such services provide, and in the absence of strong authentication at such low levels of the network (as is typical for very good reasons, see Section 1.4), the task of rebuilding the network from scratch can require a Herculean effort.

In the course of rebuilding trust in critical low-level services, having a tool that can actively probe for the presence of a malicious or compromised low-level service can help identify remnants of an attacker attempting to spoof or man-in-the-middle these services.

1.3 Focus

This paper presents an early step toward a more mature infrastructure for supporting such network recovery activities. Although we are motivated by this problem, our emphasis and focus for the scope of this paper is limited to:

- constructing a data model for representing trust relationships between network services, and;
- implementing a proof-of-concept prototype that uses packet injection (via Scapy[2]) to probe suspect services and examine their responses under the trust relationship model.

These active probes will not search for specific exploits, as the examples we discuss in Section 2.4 do. Thus, we are not aiming to create a thorough vulnerability scanner. Instead, our probes are meant to test for proper functionality and thereby trustworthiness. Our form of active probing is designed less to find out what causes a specific problem than to find whether a potential misconfiguration or malicious influence exists.

1.4 Active Intrusion Detection

Our primary contribution is to propose a new pattern for intrusion detection: actively issuing probes (in the form of specially crafted or purposefully malformed network packets) meant to reveal the presence or operation of rogue services.

Most previous work, even of an active flavor, has dealt with detecting specific vulnerabilities or exploits. In contrast, we introduce a method for crafting active scanning patterns meant to elicit a certain behavior from network hosts. Such a facility can help establish and maintain a basis for verifying the trustworthiness of network services on an ongoing basis (one can think of it as "Tripwire" for network behaviors). We are not searching for specific exploits (as the examples in Section 2.4 do), but rather search for *deception patterns* (i.e., indications that rogue services exist or that otherwise trusted services are compromised).

We believe active probing is most useful in verifying the trustworthiness of certain key network services, including DNS, DHCP, and ARP. We call these services the *Deception Surface* of the network, because it is exactly this fabric upon which most users implicitly (and often unknowingly) base their belief that they are interacting with a trustworthy network connection or service. Since these protocols rarely involve authentication, they are ripe targets for deception.

There are good reasons for *not* employing authentication and authorization infrastructure at such a low network level. The effort involved in managing this equipment and these services in the presence of a variety of different authentication mechanisms and credentials is greatly increased. Without the need to predistribute credentials, hosts are free to "plug-and-play" with the network; being able to simply trust these services by default is a labor-saving practice. For a large enterprise network, configuring each host with authentication credentials for all deception surface network services requires a large investment of valuable time and energy. Most users prefer their machines to work out of the box, and prefer to avoid extensive setup time. For this reason, such authentication (even if a mechanism exists, like DNSSEC) frequently remains unused, thereby leaving room for rogue services and deception.

Central Assumption One of the central assumptions of our approach is the hypothesis that there is an equivalence between "normal" behavior and "trustworthy" be-

havior. As a consequence, our approach is currently best suited toward detecting malicious influence or attacks that change the normal behavior of a service; in other words, we cannot detect attacks that display syntactically or semantically indistinguishable behavior (and our tool's model of a service's behavior may be incomplete, and thus unable to test features or characteristics that may have been changed). Our tool makes the assumption that changes in normal behavior are symptoms of either malicious influence or misconfiguration.

Our goal with active probing is to significantly raise the bar for an attacker: now they need not only provide a rogue service, but mimic all the logic and failure modes of the "valid" service's code logic and specific configuration. In a sense, active probing helps swing the attacker's traditionally asymmetric advantage to a network defender.

2 Related Work

Our work on active intrusion detection is inspired by recent examples of (largely manual) analysis of the properties and behavior of exploits and malware (see examples below). At the same time, the most related work from a technical perspective is the body of work on OS fingerprinting (see below) and detecting network sniffers (e.g., sniffer-detect.nse [21]). This latter script takes advantage of the fact that a network stack in promiscuous mode will pick up packets that are not intended for it, but after the stack removes the addresses, all higher layers assume the packet is intended for the local stack and act accordingly. The sniffer-detect script uses ARP probes in this manner, and by the responses it hears is able to make a determination about whether or not the probed host is in promiscuous mode. At its core, the approach exploits an assumption made within the stack: that packets which reach the upper layers of the stack are supposed to be answered by that stack. This insight is an excellent independent application of the combination of the Stimulus-Response pattern and the Cross-Layer Data pattern (see Section 3).

Port-knocking is a similar idea to active probing applied to access control: by probing a host with a particular pattern of packets, one can gain the ability to have the target firewall forward subsequent packets. In the theme of verifying host behavior to detect deviations from expected behavior, this work is conceptually related to Frias-Martinez et al. [15], who enable network access control (especially for MANET environments) based on exchanging anomaly detection byte content models.

2.1 Finding Deceptions vs. Monitoring

One central question is how much active probing differs from existing network "good hygiene" monitoring practices like using a second or third independent network connection to actively monitor properties, services, and data that your network exposes to the outside world. We note that active probing is an extension of common practice to proactively scan internal networks with tools like NMap to discover open ports, new machines, or other previously unknown activity at the network edge or within an organization's network core. Rather than just detecting open ports on machines that should not be there, our approach is predicated on reasoning about *deceptions* that exist in the network infrastructure. Although vulnerability scanning software (e.g., NeXpose[3], Nessus[4]) does probe hosts and servers, this type of probing typically focuses on identifying vulnerable versions of software services rather than detecting the presence of malcode or malicious activity.

2.2 Intrusion Detection

Network intrusion detection systems like Snort and Bro [35] have a number of advantages: since they are passive, they do not impose load on the network and they can be difficult to detect. We detail some of the differences between active and passive approaches to IDS in Table 1.

Regardless of the response mechanism or other details, an IDS usually employs a paradigm of passive monitoring which depends on tracking packet streams and delving into protocols [5]. This leaves them with several fundamental problems. IDS, whether passive or active, typically fail-open (i.e., their failure modes do not cease operation of the monitored system and they can not tell when they miss an alert, i.e., false negative).

We suggest that the most relevant shortcoming of current network IDS with respect to the concept of active probing is that an IDS is left to guess the end state of all hosts on the monitored segment. Fundamentally, IDS only observes packet flows and cannot feasibly know the end-state of every host in the network, making it susceptible to evasion attacks [30, 16]. Furthermore, trying to keep track of even limited amounts of state poses a resource exhaustion problem, and even keeping up with certain traffic loads can cause the IDS to miss packets.

2.3 OS Fingerprinting

Nmap uses a series of up to 16 carefully crafted probe packets, each of which is crafted for a variation in RFC specifications [20]. Whereas NMap issues probes to observe the characteristics of the target network stack, the p0f tool uses passive detection, and it examines various protocol fields (e.g., IP TTL, IP Don't Fragment, IP Type of Service, and TCP Window Size) [26]. Alternatively, LaPorte and Kollmann suggest using DHCP for finger-

Active	Passive
Can sound out targets	Must listen to targets
Network overhead	no network overhead
Operates noisily	Operates quietly
Minimal state storage requirement	Potentially significant storage
Creates own context	Must learn context from surroundings
Detection based on behavior	Detection based on signature and anomaly
Constant probing is noisy	Can run constantly without disturbing network
Cannot run offline	Can run offline
Can learn only what is listened for in data model	Can learn anything in a trace

Table 1: *A Comparison of Active and Passive IDS Properties*. While both approaches face some of the same challenges (e.g., being fail-open), a hybrid (tightly coupled or otherwise) approach seems promising.

printing [10], and Arkin suggests ICMP [3]. An interesting variation in this field is Xprobe2; rather than using a signature-matching approach to OS fingerprinting, it employs what its authors call a "fuzzy" approach. They argue that standard signature-matching relies too heavily on volatile specific signature elements. Xprobe2 instead uses a matrix-based fingerprint matching method based on the results of a series of different scans [4].

Fingerprinting OS network stacks and other services can be an imprecise activity frustrated by the use of virtual honeypots [29] or countermeasures like Wang's Morph (Defcon 12). Morph operates on signatures of existing production systems, rather than creating decoys. Morph scrubs and modifies inbound and outbound traffic to mimic a specific target operating system, fooling both active and passive fingerprinters [18].

2.4 Examples of an Active Pattern

The Conficker worm, unleashed in January 2009, represents one noteworthy example of malware analysis that resulted in a way to diagnose the presence of Conficker's control channel. The malware itself exploited flaws in Microsoft Windows to turn infected machines into a large-scale botnet [22]. It proved especially difficult to eradicate. Because some peer-to-peer strains of the worm used a customized command protocol, subsequent analysis and reverse-engineering provided a means of scanning for and identifying infected machines[6]. This example helps illustrate the utility of the general pattern of active probing for suspect behaviors.

The Zombie Web Server Botnet provides another example of active exploit detection. First documented in September 2009, the exploit targeted machines running web servers, and once installed set up an alternate web server on port 8080, thereby avoiding some passive IDS monitors that only watch port 80. Hidden frames on affected websites contained links pointing to free third-party domain names, which then translated into port 8080 on infected machines. These infected web servers, which also serviced legitimate sites, then attempted to upload malware and other malicious content from this rogue 8080 port [1]. If the user's web browser did not accept the uploaded malware, the exploit used an HTTP 302 Found status to redirect the user to another infected web server. From there, the exploit re-attempted the malware upload. This redirection was detectable by sending HTTP GET messages to the queried server and watching for 302 redirects [7].

As a final example of the utility of active probing, consider the Energizer DUO USB Battery Charger exploit (March 2010). The Energizer DUO Windows application allowed users to view the status of charging batteries and installed two `.dll` files, `UsbCharger.dll` in the application directory and `Arucer.dll` in the system32 directory. The software itself uses `UsbCharger.dll` to interact with the computer's USB interface, but it also executes `Arucer.dll` and configures it to start automatically.

`Arucer.dll` acts as a Trojan horse, opening an unauthorized backdoor on TCP port 7777 to allow remote users to view directories, send and receive files, and execute programs [11]. Since this rogue service responds only to outside control, passive detection may not be effective. An active probe, however, can detect the unauthorized open port even if not in use, and thus identify the infection more reliably [8].

2.5 Intrusion Recovery

Recovering a compromised host or network is a difficult task. Classic [34, 33, 9] and more recent [32, 17] accounts can both be found, but little work on systematic approaches to recovery from large scale intrusions exists [14, 25].

3 Approach

When a compromised machine exists on a network, there are two primary ways to find it. First, one can attempt to detect the malicious activity *passively*. Conventional intrusion detection systems provide a good example of this approach. However, compromised or rogue services may not display any behavior that is obviously malicious (thus evading misuse-based sensors) nor display behavior that is particularly new or different than previous packets (thus evading anomaly-based sensors).

Our approach employs *active probing*. The assumption underlying the utility of active probing is that such probing can reveal discrepancies in internal behavior or configuration — particularly at corner cases and for malformed input. In this sense, active probing helps a network defender understand how an infection alters its host's behavior or how rogue services operate.

Active probing is a suitable tool for discovery of latent or otherwise stealthy malicious influence; we can probe hosts (or the network at large using broadcast addresses) rather than waiting for them to send packets. Active probing can constructively infer network state and context by issuing targeted probes.

Active probing exploits several unique features about a networked environment; in essence, this environment represents a distributed state and a set of computations (i.e., the network stacks) involved in manipulating the global state of the network. The arrangement of these relationships and the nature of most protocol interactions provide several key areas of focus for designing probe patterns (e.g., sequences of protocol messages intended to elicit distinguishing responses).

3.1 Key Insight: Behavior Differences Due to Implementation or Configuration

During our experimentation, we frequently observed that the same stimulus produced different responses from different network entities. We discovered two reasons for this. The first reason relates to **configuration**. In some cases, responses differ because the two entities operated based on different configurations. For example, consider two identical DHCP server implementations programmed with different gateways. All other network conditions being equivalent, these two servers will always give a different result when queried, since they are programmed to do so. The richness of the configuration space can help distinguish between a rogue server set up for minimal interposition on a service and the full-featured service.

The second reason relates to **implementation**. In most cases, one or more RFCs lay out the behavior a network service or protocol should exhibit. In practice, however, we find that differences exist, whether due to lack of specification for every possible case, or simple deviance from the specification. Generally, we found that implementations perform similarly on common cases, such as well-behaved DHCP Discover packets. This observation makes intuitive sense, since specifications exist for them. It is the less well-behaved stimuli that are handled differently. Corner cases and malformed input (e.g., semantically invalid options pairings or flag settings) cause different, infrequently exercised code paths to execute – it is unlikely that an attacker has replicated such behavior with high fidelity.

Taken together, understanding these differences form the foundation of our method. If we look for both types of differences, then two entities must exhibit the exact same behaviors in order to escape notice. Put another way, if someone wants to masquerade as another on the network, the imitator must mimic not just the target's normal behaviors in common cases (relatively easy) but the minor, idiosyncratic ones as well (we claim that this is harder).

3.2 Stimulus-Response Pattern

We note that many network interactions take the form of pairing between stimulus and response. The DHCP Discover/Response cycle, the DNS Query/Response cycle, and many others all fall into this category, whereas something like the Cisco Discovery Protocol does not. Note that the stimulus-response includes not only client-server interactions, but also peer-to-peer as well. We rely on and harness this stimulus-response paradigm for our verification method.

3.3 Network Trust Relationships and Trusted Data

Trust relationships form the basic building block of the network. In the majority of cases, hosts trust essential services by default, to ensure ease of connection without the burden of extensive configuration. As an example, without prior configuration in an IPv4 environment, DHCP and ARP provide the primary ways for a host to learn about the network. Unfortunately, the scope of many modern networks makes these trust-by-default relationships all but necessary, since manually configuring and re-configuring every host in the network is often impractical. As a consequence, they present an avenue for an adversary who can masquerade as a provider of one of these legitimate trusted services. If the adversary offers the same trusted-by-default service and can get his or her information believed, then he or she has compromised whatever elements of the network believe that information. We target this sort of "trusted–by–default"

deception.

Note that we do not specify anything about the exact process by which the deception we have just described is executed. It could be that the adversary has disabled the legitimate service, or is simply able to get its information out faster than the legitimate information does. Regardless of the specifics, we begin in the place of a network entity and mistrust the service provider that we hear but whose trustworthiness we must accept for normal operation. Everything we do constitutes an attempt to verify the trustworthiness of that service provider. Is the information they provide consistent? Do they respond in the way a legitimate service might if we make illogical or semantically invalid requests? Or, if they are an attacker intent on remaining stealthy, do they greedily respond to packets that look attractive to intercept and interpret, but are really meaningless (in terms of us getting on the network) and only mean to flush out such malicious interposition?

3.4 Cross-Layer Data

Sometimes, it helps to exploit the layered nature of network protocols. Consider a man in the middle attack, one of the most basic and most common compromises. An ordinary machine will pick up all packets and examine them, discarding any that are not addressed to it. This behavior is expected from the majority of well-behaved machines on a network. However, a machine acting as a MITM will pick up these packets, examine them, perform some sort of malicious activity (be it recording, modifying, fuzzing, or any number of other things), and then send them on to their destination. To do this, the attacker must modify the machine's normal network stack, and configure the kernel to forward packets. This modification makes the compromise detectable (see Section 6 for our experiment on this topic).

4 Active Probing Model

We model active probing on the concept of a network conversation containing messages that reveal the violation of conditions related to configuration or behavior, where these constraints represent the belief of the probing entity about the valid, trustworthy state of the network.

In essence, active probes attempt to verify some behavior of the target host or service, and the messages emitted from the target host in response to our (crafted) protocol messages represent characteristics of that behavior. Figure 1 depicts this interplay in a very basic form; the intent behind probing is to discover behavioral artifacts arising from differences in *implementation* or *configuration* (as discussed in Section 3.1).

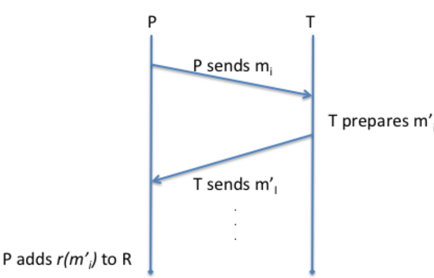


Figure 1: *Ladder Diagram for Active Probing Data Model*. A probing host P (our prototype plays this role) issues probes designed to exercise logic and configuration corner cases in the target host T. As T reacts to these probes (and generates m_i' subject to its implementation quirks and configuration details), P builds a set of data relevant to the trust relationship being probed.

Our model consists of two parties P and T. P is the prober and has the ability to simulate multiple protocol stack implementations (especially "broken" ones). The second, T, is the target or service provider. P's hypothesis is that T may contain a broken, partial, incomplete, or incorrectly configured protocol stack. If T were a trustworthy service, it would display "normal" expected behavior according to the trust relationship between P (rather, the role of the client or peer that P is playing) and T (more specifically, the server or peer that T may be masquerading as). In this sense of having an established trust relationship, we say that T provides a service X to P.

For P to consider T trustworthy with respect to service X, T must satisfy a set of conditions C on its behavior. To verify that these constraints hold, P uses a sequences of messages $M = m_1, m_2, \ldots, m_n$ sent to T that take the form of packet probes.

For each m_i, there exists a corresponding message m_i' from T to P that may be a packet, a sequence of packets, or the absence of a packet (determined through a preconfigured timeout). For each such m_i', there is some relevant portion $r(m_i')$ that serves as evidence for or against some particular element $c_i \in C$. As each m_i' is received (or not received), P performs the operation $R = R \cup r(m_i')$, building a body of evidence R as shown in Figure 1. Once all probes have been sent and answers recorded, the probing entity decides whether or not R violates the conditions contained in C.

5 Methodology

In attempting to recover trust in key network services, a system administrator would follow the general tasks outlined below. The procedures we describe are typically aimed at an auditing-style service rather than a general-purpose scanner for running malware or botnet command and control. As such, the definition of a trust relationship, the specific verification plan, and the format and content of probes are usually service specific and informed by administrator knowledge of their own service implementations and configuration. We posit (but have not shown) that the amount of effort needed to follow the steps below is similar to tuning of IDS rules or calibration of IDS parameters to a specific environment.

5.1 Define Trustworthiness

In order to establish the trustworthiness of a network service, there must be a notion of what trustworthiness means for that service. This will vary based on the network and service being verified, and in most cases will depend on the specific deployment of the service being probed. This trustworthiness criteria directly informs the set of constraints C.

For example, trustworthiness in the forwarding case means that no hosts but known gateways should exhibit forwarding behavior. For a more complicated system, a definition might take into account information the legitimate service should provide (for example, a known-valid set of DNS responses), and ways it should respond to certain stimuli (e.g., how the service handles a particular corner case configuration or incompatible flags). Generally speaking, the definition is what we need to hear to trust the speaker.

5.2 Verification Plan

Once we have an idea of what trustworthiness looks like, we need to develop a plan of how to verify it. Recall the two types of differences between service providers we discussed earlier. Many network entities have peculiarities to their implementations, and the plan for verification should make use of them. It is also necessary to get as much standard information from the probed entity, so that both types of differences can be detected. The more information gathered, both about the service implementation and the service configuration, the harder an adversary must work to fool our probe. In doing this we need to plan to check our service against every part of the trustworthiness definition we have already developed.

For the examples above, the verification plan might range from a simple comparison of known good answers to specific DNS queries to the absence of a "forwarded" packet. In essence, each verification plan is tightly coupled to the actual method of detecting a specific deception on the network. As yet, we do not contend with automating this process.

5.3 Probe Creation

The next step in our methodology calls for turning the plan into a set of active message probes and codifying those probes. Although a variety of packet crafting mechanisms exist, we found Scapy, a packet generation and manipulation tool, to be helpful. The codified probes crafted in Scapy's environment comprise the functional portion of an active verification tool.

5.4 Reply Detection

Finally, we need to capture the replies to our probes and examine them against the constraints derived from our trustworthiness definition. With that information, we must make a determination as to the trustworthiness of what responses the probes cause.

5.5 Implementation

We have found Scapy [27], a freeware packet manipulation program, quite useful. Scapy allows users to build, sniff, analyze, decode, send, and receive packets with incredible flexibility. It does not interpret response packets directly, so it can prove more useful than other packet injection or scanning tools in some scenarios. It employs Python-based control, so its commands are also easily adapted into Python programs. We have used Scapy to implement our prototype probing tool. Currently, verification plans (and their corresponding probes) require individually-developed Scapy scripts.

6 Case Studies: Detecting Deceptions

Our preliminary evaluation focuses on illustrating our prototype's effectiveness at detecting network deceptions rather than attempting to detect malicious software (e.g., botnet command-and-control, spyware). To a certain extent, the related work we discuss in Section 2 illustrates how one might go about using existing tools like nmap to identify command-and-control or backdoors. Although we illustrate how to detect (1) a duplicate DHCP server and (2) the presence of a host configured for forwarding, our point is that these two examples are patterns of network deceptions, and this is the main intent of our approach.

6.1 Detecting Forwarding Behavior

As an example of using the Cross-Layer Data pattern, one of our first experiments dealt with detection of forwarding behavior. As ordinary machine will typically silently discard packets (frames) not addressed to it. This behavior is expected from the majority of well-behaved machines on a network. A machine acting as a MITM, however, will pick up these packets, examine them, perform some sort of malicious activity (be it recording, modifying, fuzzing, or any number of other things), and then send them on to their destination. To accomplish this MITM, the attacker must modify the machine's normal network stack settings and configure the kernel to forward packets. This modification is remotely detectable.

We hypothesized that if we sent a broadcast packet out to the network with the destination as our own machine, a host configured for forwarding might give itself away by sending the packet back to us. We used Scapy to test this, sending IP packets carrying a layer 2 broadcast address and a layer 3 address of our own machine. We found that many forwarding entities (for example, Linksys routers) did identify themselves by forwarding the packet as expected, but Linux kernels in forwarding mode do not. We hypothesized that this was due to the layer 2 broadcast address of the packet. To test this hypothesis, we replaced the broadcast hardware address with a unicast address of the machine we wanted to probe, and listened for the response. We found that this resulted in the packet being sent back to us, as expected. We codified this result into an Nmap plugin that detects hosts in forwarding mode that are behaving in what is generally an undesirable manner and thus *may* have been compromised or misconfigured.

Formally speaking, in this experiment of detecting a host in forwarding mode, the condition C is that only a small known set of hosts on the network should be in forwarding mode (specifically: that only those hosts should deliver the packet we generated back to us because we chose the packet contents in such a way as to be consumed by hosts that are promiscuous and forwarding, but when processed by higher layers of the network stack, don't realize that they shouldn't be sending this packet back to its origin); if the responses that P gathers contains an IP address outside this set (i.e., we see our message from m_i in R), we know that the trust condition is violated.

6.2 Rogue DHCP Server

To demonstrate the viability of an active probing approach, we have implemented it on the Dynamic Host Configuration Protocol. DHCP makes an excellent subject for a case study for several reasons. First, it provides new hosts several critical pieces of knowledge about the network, such as an IP address, gateway information, and location of the DNS servers. Typically, a network stack sends out a DHCP Discover immediately after coming online, highlighting DHCP's importance. If an adversary can get malicious DHCP information believed, he or she can exert a great deal of control over the deceived hosts. Second, it comprises part of our Deception surface, so most hosts trust whatever DHCP traffic they receive by default.[5]

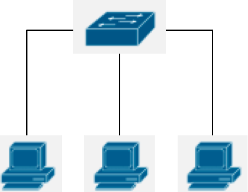


Figure 2: *Basic DHCP Setup With Cisco Switch*. This self-contained test environment consists of a set of computers connected to a single Cisco switch (a DHCP server, the rogue DHCP service, and our prober P). This setup was also used for the forwarding detection scenario.

We see a recent example of an exploit using DHCP in a variant of the Alureon rootkit. This exploit infects networks and sets up a rogue DHCP server to compete with the legitimate one. This rogue server gives out the address of a DNS server under the control of the worm's authors, which in turn points users to a malicious web server. This web server attempts to force the user to update their browser, but they instead are downloading a malware that will reset their DNS pointer to Google's service once the machine is infected [2]. This is the sort of exploit that motivates us to examine DHCP closely.

Our prototype software uses Scapy scripts to probe DHCP servers and can both produce `PCAP` fingerprint files and compare to an existing `PCAP` fingerprint. In practice, we found it successfully distinguished between the different servers we used.

6.3 Environment

We used three main environments for our DHCP experiments. First, as shown in Figure 2, we have a small, self-contained test environment consisting of a set of computers connected to a single Cisco switch. This was also the environment we used for the aforementioned forwarding detection work. We configured one of the computers with an instance of the udhcpd DHCP server, and ran our tests from the other.

We also have access to the production network at Dartmouth College's CS department. We used the same machine for our tests here as in the previous environment, and ran our experiments against the actual production server. Finally, we have the DHCP server included in a Linksys WRT54G2 router. It runs as the core of a small home network.

As described previously, we tried to determine both the configuration of the probed server and the server itself. We are not interested in identifying the specific server implementation, but rather detecting its differences from another server. We do so by taking a brute-force approach, where we try several different values for different fields. Some of these are well-behaved values, and others are designed to test the server's handling of unusual traffic. This allows us to test both the server's configuration and its implementation.

To test the usefulness of our software, we compared responses to probes across our test environments. Doing so simulates the introduction of another DHCP agent onto the network whose traffic we are seeing instead of the legitimate server's traffic. This process is akin to comparing a previous behavior model captured in a known trustworthy state with a later behavior model gleaned from an environment during recovery. We can probe the server at a time when we assume it to be in a trustworthy state; this can be established by (1) manual inspection of the program or process, (2) some kind of integrity check of the code and configuration files a la Tripwire, or (3) immediately after a new deployment of the service.

6.4 Constructing DHCP Probes

Within a DHCP packet (see Figure 4), four fields (ciaddr, yiaddr, siaddr, and giaddr) contain IP addresses, while a fifth (chaddr) contains a MAC address. We check the servers' handling of these fields by setting each one in turn to four different types of values:

- The client's currently assigned IP address
- Another valid IP address in the client's subnet
- A valid IP address in another subnet
- An invalid IP address

We also do something similar for the chaddr field:

- The client's MAC address
- Another valid MAC address
- An all-zeroes MAC address
- An all-ones (Broadcast) MAC address

```
# Request probe w/ ciaddr set to other IP
state = random.getstate()
probeFunc(Ether(src=get_if_raw_hwaddr(conf.iface)[1],
                dst="ff:ff:ff:ff:ff:ff")
          /IP(src="0.0.0.0", dst="255.255.255.255")
          /UDP(sport=68, dport=67)
          /BOOTP(flags=0x8000,
                 chaddr=get_if_raw_hwaddr(conf.iface)[1],
                 giaddr=ip,
                 xid=random.randint(0, 4294967295))
          /DHCP(options=[("message-type", "discover"),
                         ("end")]
               ), state)

# Request probe w/ chaddr zeroes
state = random.getstate()
probeFunc(Ether(src=get_if_raw_hwaddr(conf.iface)[1],
                dst="ff:ff:ff:ff:ff:ff")
          /IP(src="0.0.0.0", dst="255.255.255.255")
          /UDP(sport=68, dport=67)
          /BOOTP(flags=0x8000,
                 chaddr="00:00:00:00:00:00",
                 xid=random.randint(0, 4294967295))
          /DHCP(options=[("message-type", "discover"),
                         ("end")]
               ), state)

# Request probe with chaddr nonsense
state = random.getstate()
probeFunc(Ether(src=get_if_raw_hwaddr(conf.iface)[1],
                dst=''ff:ff:ff:ff:ff:ff'')
          /IP(src=''0.0.0.0'', dst=''255.255.255.255'')
          /UDP(sport=68, dport=67)
          /BOOTP(flags=0x8000,
                 chaddr=''gg:gg:gg:gg:gg'',
                 xid=random.randint(0, 4294967295))
         /DHCP(options=[(``message-type'', ``discover''),
                        (``end'')]
              ), state)
```

Figure 3: *Example Probes for DHCP*. Three of the eleven probes we constructed for profiling the behavior of a DHCP server. We took a profile of the known good DHCP service and compared it against another profile from a different machine.

We also send discover probes that manipulate option values. These include normal options and the parameter request list, which allows the requesting client to ask for specific information from the server. We set a number of options in our probes and assign them values (where applicable) as described above. We also send a number of probes requesting different information from the server using the parameter request list option (we do not believe that Scapy implements all of the options).

6.5 Results

The tool successfully identified that significant differences exist between the production DHCP server and the Linksys router. Not only were the configurations dif-

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     op (1)    |   htype (1)   |   hlen (1)    |   hops (1)    |
+---------------+---------------+---------------+---------------+
|                            xid (4)                            |
+-------------------------------+-------------------------------+
|           secs (2)            |           flags (2)           |
+-------------------------------+-------------------------------+
|                          ciaddr  (4)                          |
+---------------------------------------------------------------+
|                          yiaddr  (4)                          |
+---------------------------------------------------------------+
|                          siaddr  (4)                          |
+---------------------------------------------------------------+
|                          giaddr  (4)                          |
+---------------------------------------------------------------+
|                                                               |
|                          chaddr  (16)                         |
|                                                               |
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                          sname   (64)                         |
+---------------------------------------------------------------+
|                                                               |
|                          file    (128)                        |
+---------------------------------------------------------------+
|                                                               |
|                          options (variable)                   |
+---------------------------------------------------------------+
```

Figure 4: *DHCP Message Format*. This diagram was copied verbatim from RFC2131 [13].

ferent, but it turned out that the Linksys DHCP agent in our third environment ignored several of the less well-behaved probes, leading to an easy identification. While not comprehensive, we believe this successful result demonstrates the value of our approach to active intrusion detection.

7 Discussion & Future Work

We discuss how our active probing methodology would apply to two other critical network services (DNS and ARP). This is in essence future work, but we offer the sketches as evidence of the feasibility of extending this type of probing to other fundamental network services. We are currently extending our analysis (and crafting probes) to other services like SNMP, STP, NTP, and routing protocols. Each of these protocols requires a different type of approach to composing a verification plan, since their modes of operation may not naturally fit a query-response pattern. In such scenarios, we can take advantage of the cross-layer data pattern and trust relationship patterns.

In the two examples below, our probing has different semantics than our "rogue DHCP" and "forwarding detection" examples. Since we sketch an outline of a verification plan, we focused on relatively easy ways to verify the trustworthiness of these services (e.g., for DNS comparing against known good answers). We could, however, rely on a behavioral signature much more in line with the DHCP experiment by issuing probes that exercise little-used options or ask for incomplete or illogical DNS and ARP resolutions.

7.1 Domain Name System

We describe one way in which our method might apply to Domain Name System (DNS). DNS operates on the stimulus-response client-server model, where the client sends name resolution requests to the server, which in turn queries as many other servers in the DNS hierarchy as is necessary to get an answer [23, 24]. In many cases, DNS responses are trusted by default, since they represent the best and frequently only information a host has about how to resolve external names to machine addresses. As such, attacks on DNS are fairly common, since successfully doing so could trick a host into sending all of its traffic to the adversary.

Clearly, the trustworthiness of DNS depends on giving correct answers to queries. Our system should be able to determine whether or not the responses it hears are correct. If not, we can assume that the server we are querying is untrustworthy. Note that this does not mean that the server we are querying directly has an issue, but since DNS servers form a hierarchy, a trust issue with one could mean trouble for many others.

In creating a verification plan, we cannot feasibly examine what answers the server gives for every possible query. We can, however, pick a number of common queries and build a list of responses we should receive for each one. We need a list rather than a single response, as one name frequently has several hosts which respond to queries for it. This list should be large and diverse, and the answers built from manual research or compiled from DNS queries to different servers, minimizing the possibility that a compromised server contributes to our definition of trustworthiness. We also want to feed the server some malformed requests, both with poorly-formed packets and for names known not to exist (this will have to be checked) to test the implementation details of the server.

Our probes would take the form of DNS question packets as described previously, which could be done with Scapy. Query responses could be listened for, and responses checked against the list discussed previously. If we hear any unexpected responses, an alert could be raised indicating that a possible issue with the server exists.

7.2 Address Resolution Protocol

We describe how our method could apply to Address Resolution Protocol (ARP). ARP operates on the stimulus-response model where each host or gateway can both make and service requests [28]. ARP provides important information enabling communication both within and across networks, and its information is generally trusted by default, so it provides a good illustration for

Protocol	P	T	X
DNS	Host	DNS Server	Name resolution information
DHCP	Host	DHCP Server	IP address, gateway address, DNS address, etc.
ARP	Host/Gateway	Host/Gateway	IP address:MAC address mapping
STP	Switch	Switch	Bridge priority, path cost
CDP	Network Device	Network Device	Addresses, device information

Table 2: *Enumerating Services Involved in a Network's "Deception Surface"*. The protocols here form the main deception surface of a network; we list them in the context of our data model's trust relationship syntax. Even though there are "secure" variants of some of these protocols, networks do not always use them because requiring authentication infrastructure in order to establish basic layer 2 and 3 connectivity can be cumbersome and difficult to maintain.

active probing.

The definition of trustworthiness for ARP should state that all hosts respond to queries for their IP address with their own MAC address, and gateways also respond to queries for IP addresses outside their network segment with their own MAC address. Any deviance from this model could indicate a deception occurring.

Merely looking at ARP replies in isolation may not be sufficient. Consider the following scanning strategy. A prober conducts an ARP scan of a given set of addresses, and for each address scanned it does two things. First, it listens for replies and raises an alert if it hears more than one different MAC address in response. Second, if only one response is received, it saves that response to a hashtable. It then would check for one of two conditions. The scan can either look for the address it has just heard appearing in the hashtable twice, or to look for it to not be in the hash.

The prober needs to run this scan against both its local network (excluding the gateway) and against addresses outside its network. The former warns the prober of untrustworthy ARP behavior of hosts and servers on its own segment, and the latter of such behavior associated with its gateway. The scan needs to look for both the presence of duplicate addresses and their absence for this reason: all non-local addresses should resolve to the same address, which should not have been seen for any local address.

If we do not observe this, we know that we have traffic intended for multiple IP addresses going to the same device on the network. This falls outside the definition of trustworthy ARP behavior, and the prober can raise an alert. We could run forwarding detection against the non-gateway IPs which returned the duplicate MAC, but it is not strictly necessary. Probes would take the form of a simple ARP scan, with a supporting hashtable. The technique employs a brute-force approach, but should successfully detect ARP issues on the network.

7.3 Limitations

Although the probing approach we discuss is meant to serve as a kind of "tripwire for trust," it has several shortcomings. Of particular interest going forward is the consideration of how to scale the process of producing a verification plan and the concomitant probes to very large networks (along with large networks containing non-TCP/IP networking equipment). In a sense, the manual nature of writing probing scripts both helps and hinders the ability to scale. On one hand, writing scripts for a small number of critical pieces of network infrastructure benefits from the manual attention to detail and the knowledge of the system administrator about the quirks or peculiarities of the system being probed. On the other hand, in a highly heterogeneous environment containing a network composed over years from a variety of organizations, the sheer diversity of core services poses a significant challenge.

One way to deal with this challenge is to focus on detecting the presence of certain types of deceptions rather than verifying the behavior of every last system. Another (complementary) approach would require research that can attempt to generate a set of probes from pristine (or trusted) configuration files and/or binary code of the target service.

The stimulus-response pattern for detecting untrustworthy behavior may not apply well to protocols that are not purely request-response based (e.g., they may operate on a stream of asynchronous update messages). We can attempt to verify the behavior of such services through trust relationships and cross-layer data (for example, for a routing protocol we might spoof or issue route withdrawals or announcements from one peer and see if the target announces such messages to another peer).

Finally, we have not explicitly considered the effect the use of such active probing might have on IDSs extant in the target environment. It is likely that certain types of IDS might alert on messages from the prober, especially if they are malformed in some fashion. Dangers here include the IDS increasing its alert logging (and thereby

increasing the noise in its alert stream or logs) as well as subtly changing their view of the network. In general, a coordinated security response from multiple independent security mechanisms is a hard unsolved problem. Nevertheless, one of our primary use cases is in a network that we are attempting to recover; we might expect to ignore the secondary effects of such probing in favor of re-establishing core critical services.

8 Conclusion

This paper suggests that active intrusion detection (AID) techniques hold promise as a useful network security pattern, particularly when attempting to verify that basic constraints or characteristics of the network hold true. We presented an approach to AID based on probing: issuing crafted packets meant to elicit a particular type of response from the target system or host.

There are several conceptual lessons to take away from this work. Our main approach is predicated on probing the "corner case" behavior and configurations of network services and verifying that services return known–good answers. Our main assumption is that normal behavior is in some sense equivalent to "trustworthy." Feeding a system crafted input meant to exercise corner cases in logic or configuration serves as a good heuristic for revealing behavior that might carry highly individualized information. We hypothesize that meaningful differences in the characteristics of network trust relationships can reveal malicious influence (or at least a bug or misconfiguration).

We suggested three patterns for building verification plans and exploring this space of varied behavior: stimulus-response, cross-layer data, and trust relationships. This approach can help users, client hosts, and system administrators verify the trustworthiness of network services, especially in the absence of strong authentication mechanisms at layer 2 and 3. We discussed how to apply this method to DNS and ARP, we crafted packets that can remotely detect a host in forwarding mode, and we implemented a Scapy-based prototype to verify the trustworthiness of a DHCP service.

9 Acknowledgments

We appreciate the insight and comments of the LISA reviewers. In particular, they asked us to provide more detail on our prototype and data model as well as more carefully discuss the current limitations. Our shepherd, Tim Nelson, showed a lot of patience in working with us to reconcile some of the submission manuscript's shortcomings; the final paper is much improved because of his input and guidance.

Locasto is supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Dynamic DNS and Botnet of Zombie Web Servers. *Unmask Parasites.blog* (September 11, 2009). http://blog.unmaskparasites.com/2009/09/11/dynamic-dns-and-botnet-of-zombie-web-servers/.

[2] Worm uses built-in DHCP server to spread. *The H: Security News and Open Source Developments* (2011). http://www.h-online.com/security/news/item/Worm-uses-built-in-DHCP-server-to-spread-1255388.html.

[3] ARKIN, O. ICMP Usage in Scanning or Understanding some of the ICMP Protocol's Hazards. Tech. rep., The Sys-Security Group, December 2000.

[4] ARKIN, O., AND YAROCHKIN, F. Xprobe v2.0: A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. Tech. rep., August 2002. http://ofirarkin.files.wordpress.com/2008/11/xprobe2.pdf,.

[5] AXELSSON, S. Intrusion Detection Systems: A Survey and Taxonomy. Tech. rep., Chalmers University of Technology, 2000.

[6] BOWES, R. Scanning for Conficker's peer to peer. *Skull Security* (April 25, 2005). http://www.skullsecurity.org/blog/2009/scanning-for-confickers-peer-to-peer.

[7] BOWES, R. Zombie Web servers: are you one? *Skull Security* (September 11, 2009). http://www.skullsecurity.org/blog/2009/zombie-web-servers-are-you-one.

[8] BOWES, R. Using nmap to detect the arucer (ie, energizer) trojan. *Skull Security* (March 8, 2010). http://www.skullsecurity.org/blog/2010/using-nmap-to-detect-the-arucer-ie-energizer-trojan.

[9] CHESWICK, B. An Evening with Berferd, in which a cracker is lured, endured, and studied. In *Proceedings of the Winter USENIX Conference* (January 1992).

[10] DAVID LAPORTE AND ERIC KOLLMANN. Using DHCP for Passive OS Identification. BlackHat Japan.

[11] DORMANN, W. Vulnerability note vu#154421. *US-Cert Vulnerability Notes Database* (March 5, 2005). http://www.kb.cert.org/vuls/id/154421.

[12] DREGER, H., FELDMANN, A., MAI, M., PAXSON, V., AND SOMMER, R. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *Proceedings of the USENIX Security Symposium*.

[13] DROMS, R. Dynamic Host Configuration Protocol, March 1997. http://www.ietf.org/rfc/rfc2131.txt.

[14] DUNLAP, G. W., KING, S., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)* (February 2002).

[15] FRIAS-MARTINEZ, V., SHERRICK, J., D.KEROMYTIS, A., AND STOLFO, S. J. A Network Access Control Mechanism Based on Behavior Profiles. In *Proceedings of the Annual Computer Security Applications Conference* (December 2009).

[16] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *USENIX Security Symposium* (2001).

[17] HILZINGER, M. Fedora: Chronicle of a Server Break-in. http://www.linux-magazine.com/linux_magazine_com/online/news/update_fedora_chronicle_of_a_server_break_in, March 2009. Linux Magazine.

[18] KATHY WANG. Frustrating OS Fingerprinting with Morph. In *Proceedings of DEFCON 12* (July 2004). http://www.windowsecurity.com/uplarticle/1/ICMP_Scanning_v2.5.pdf.

[19] LOCASTO, M. E., BURNSIDE, M., AND BETHEA, D. Pushing boulders uphill: the difficulty of network intrusion recovery. In *Proceedings of the 23rd conference on Large installation system administration* (Berkeley, CA, USA, Nov 2009), LISA'09, USENIX Association.

[20] LYON, G. *Remote OS Detection*, nmap reference guide, chapter 8 ed., 2010.

[21] MAREK MAJKOWSKI. sniffer-detect.nse. Nmap Scripting Engine Documentation Portal. http://nmap.org/nsedoc/scripts/sniffer-detect.html.

[22] MARKOFF, J. Worm infects millions of computers worldwide. *The New York Times* (January 22, 2009).

[23] MOCKAPETRIS, P. RFC 1034: Domain Names - Concepts and Facilities, 1987. http://www.ietf.org/rfc/rfc1034.txt.

[24] MOCKAPETRIS, P. RFC 1035: Domain Names - Implementation and Specification, 1987. http://www.ietf.org/rfc/rfc1035.txt.

[25] OZGIT, A., DAYIOGLU, B., ANUK, E., KANBUR, I., ALPTEKIN, O., AND ERMIS, U. Design of a log server for distributed and large-scale server environments.

[26] PETERSEN, B. Intrusion Detection FAQ: What is p0f and what does it do? Tech. rep., The SANS Institute. http://www.sans.org/security-resources/idfaq/p0f.php.

[27] PHILIPPE BIONDI. *Scapy v2.1.1-dev documentation*, April 19, 2010. http://www.secdev.org/projects/scapy/doc/.

[28] PLUMMER, D. C. RFC 826: An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware, 1982. http://tools.ietf.org/html/rfc826.

[29] PROVOS, N. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 1–14.

[30] PTACEK, T. H., AND NEWSHAM, T. N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Tech. rep., Secure Networks, Inc., January 1998.

[31] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of* 13th *LISA Conference* (November 1999), pp. 229–238. http://www.usenix.org/event/lisa99/roesch.html.

[32] SINGER, A. Tempting Fate. *USENIX login; 30*, 1 (February 2005), 27–30.

[33] SPAFFORD, E. H. The Internet Worm: Crisis and Aftermath. *Communications of the ACM 32*, 6 (June 1989), 678–687.

[34] STOLL, C. Stalking the Wily Hacker. *Communications of the ACM 31*, 5 (May 1988), 484.

[35] VERN PAXSON. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium* (Berkeley, CA, USA, 1998), USENIX Association. http://www.usenix.org/publications/library/proceedings/sec98/paxson.html.

Notes

[1] http://www.h-online.com/security/news/item/Worm-uses-built-in-DHCP-server-to-spread-1255388.html

[2] http://www.secdev.org/projects/scapy/

[3] http://www.rapid7.com/products/nexpose-community-edition.jsp

[4] http://www.tenable.com/products/nessus

[5] Although DHCP in IPv6 operates slightly differently, opportunities for masquerading still exist, and active probing can help detect such attacks.

Community-based analysis of netflow for early detection of security incidents

Stefan Weigert[†] Matti A. Hiltunen[‡] Christof Fetzer[†]

[†]TU Dresden
Dresden, Germany
{stefan,christof}@se.inf.tu-dresden.de

[‡]AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
hiltunen@research.att.com

***Abstract*—Detection and remediation of security incidents (e.g., attacks, compromised machines, policy violations) is an increasingly important task of system administrators. While numerous tools and techniques are available (e.g., Snort, nmap, netflow), novel attacks and low-grade events may still be hard to detect in a timely manner. In this paper, we present a novel approach for detecting stealthy, low-grade security incidents by utilizing information across a community of organizations (e.g., banking industry, energy generation and distribution industry, governmental organizations in a specific country, etc). The approach uses netflow, a commonly available non-intrusive data source, analyzes communication to/from the community, and alerts the community members when suspicious activity is detected. A community-based detection has the ability to detect incidents that would fall below local detection thresholds while maintaining the number of alerts at a manageable level for each day.**

I. Introduction

Detection and remediation of security incidents (e.g., attacks, compromised machines, policy violations) is an increasingly important task of system administrators. While numerous tools and techniques are available, novel attacks and low-grade security events may still be hard to detect in a timely manner. Specifically, system administrators typically have to base their actions on observing the local traffic to and from their own networks as well as global security incident alerts from organizations such as SEI CERT[1], Arbor Atlas[2], or software and hardware vendors. However, stealthy targeted attacks may slip below detection thresholds both in the local data alone or on the global scale.

Furthermore, the nature of internet-based attacks is changing from random hacking to financially or politically motivated attacks. For example, botnets are increasingly leased out to highest bidders and DDoS attacks are often used as a means for blackmail. Moreover, attacks targeting industries with financial information (e-commerce, banking, gaming, insurance) are increasing and the threat of attacks against SCADA (supervisory control and data acquisition) systems in electrical power generation, transmission, and distribution (among other industrial process control systems) is even considered a potential target for terrorism [10].

[1]http://www.cert.org/
[2]http://atlas.arbor.net/

Targeted attacks might not leave a large traffic footprint in the targeted organization since one machine with access to the desired information or control system may be sufficient for the attacker to achieve their goals. It is often difficult to detect such low-footprint attacks based on local monitoring alone because it is often necessary to set local alerting thresholds high enough not to generate too many false positives and overwhelm the system administrators. But as a result, a stealthy attack or compromise may lay undetected. Therefore, it is possible for an attacker to target many such organizations without being detected. For example, the attacker may want to maximize profit by attacking multiple financial organizations concurrently before the vulnerability used is detected and corrected. Similarly, terrorists may require the control of many companies to achieve their goal of large scale damage.

In this paper, we present a novel approach for detecting stealthy, low-grade security incidents by utilizing information across a community of organizations (e.g., banking industry, energy generation and distribution industry). We will show by using an example that we can find possible attacks (or attempts) that only transfer very little data (e.g., a few bytes) and thus would remain undetected by conventional approaches.

The remainder of this paper is structured as follows. In Section II, we present the technical approach based on netflow data and construction of communities of interest. Section III describes the implementation of the system, including the algorithms used for the analysis. We evaluate the performance of our system in Section IVi and present selected case studies of suspicious activity we have identified in Section V . Section VI outlines related work in the area and Section VII concludes the paper.

II. Approach

A. Service vision

Our technique is based on the concept of *community*, in our case defined as a collection of (at least two) organizations. A community can be specified based on any criteria relevant for attack detection. For example, it could consists of businesses in a particular industry (e.g., banking, health care, insurance, etc), organizations within a country (e.g., businesses and government agencies in one country), or organizations with particular

type of valuable information (e.g., industrial espionage or customer credit card information). We detect stealthy security attacks by observing the communication to/from the member organizations of a community. The intuition being that within each organization only very few machines may be attacked or compromised and as a result an attack can be very hard to detect within each organization. However, by observing the communication behavior across multiple organizations in the community, such stealthy behavior may become visible.

Given that we analyze communication in the Internet, each organization is defined by the list or range of IP addresses belonging to the organization. We consider Internet communication connections (reported by netflow, for example) within the communities and between communities and external IP addresses who do not belong to any community. For our analysis, all the IP addresses within an organization can be collapsed into one identifier representing the organization. Any communication between two IP addresses where neither belongs to one of our communities and neither has communicated with a community in the past can be ignored. Furthermore, communication with IP addresses belonging to commonly used Internet services (e.g., search, news, social media) can be white listed and removed from consideration.

We construct a communication graph for each IP address that communicates with at least one organization in a community as illustrated in Figure 1. This figure shows the communication graph for an external IP address (i.e., some IP address outside any of the communities of interest). This node has communicated with two communities, one consisting of organizations 7 and 8, and the other consisting of organizations 1 through 6. A directed edge from some node A to some other node B in the graph indicates that A has sent messages to B. Although not depicted in the figure, each edge may contain additional information, such as the combinations of source and destination ports used.

The weight of the edge is used to quantify the importance of the communication. The importance can be based simply on the number of messages or bytes sent, or the number of contacted individual members in the targeted organization. However, some communication may be more important than others from security point of view. For example, some port numbers are more often involved with malicious activity (e.g., based on CERT reports) and communication using such ports can be weighted more heavily.

The weight is also used to limit the size of each graph. The size of the graph is determined by the number of nodes it contains. If the size exceeds a given threshold, we remove the weakest links until the threshold is reached. This is necessary because storing all communications would require too much space even for a single day. For example, in our data set consisting of heavily sampled netflow, a given weekday contains about 860 million entries. These 860 million recorded netflows originate in 28 million distinct IP addresses. Therefore, if we would not filter unimportant IP addresses, we would need to store 28 million graphs. Moreover, each of these 28 million IP addresses often connects with 1 to 2 million other IP addresses.

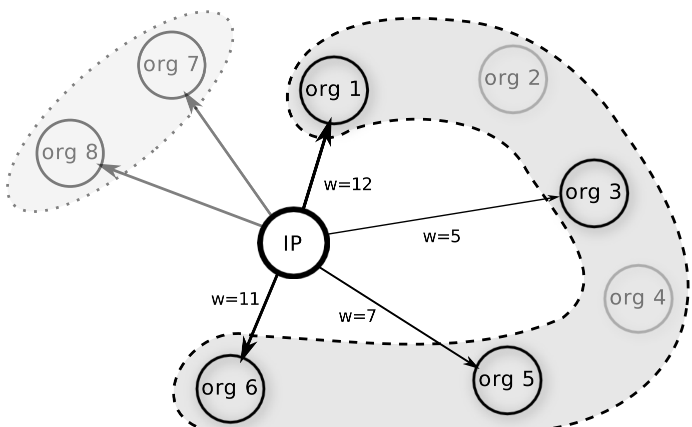


Fig. 1: Communication graph for an IP address

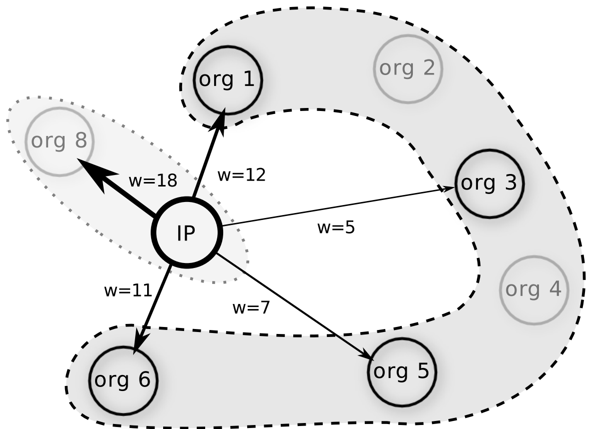


Fig. 2: Communication graph for a community member

Thus, if we did not limit the size of each graph, we would have some graphs that are too large to fit into memory. The situation would be even more challenging if we analyzed the data for one month or a week instead of the current one day at a time.

As already stated, we also consider communication within a community and across communities. With that, we are able to detect already compromised computers inside an organization when they try to attack further organizations as shown in Figure 2. To reduce the number of false positives (many organizations have frequent contact with other organizations of the same or other communities), a computer inside an organization that belongs to a community (or is contained in the whitelist) has to show more suspicious behavior than an external IP address before an alarm is generated. For example, we do not consider communication via port 443 with or across communities.

Given such communication graphs, a potential security incident is suspected when an IP address communicates with a specified number of community members. Typical examples of security threats that can be detected using this approach include botnet controllers managing a number of bots in the

community, compromised machines downloading stolen information on a dedicated server, an attacker targeting machines in multiple organizations, as well as many security policy violations (e.g., illegal software download sites, etc). The number of alarms can be controlled using thresholds and the system can memorize IP addresses that have already been reported recently. When there are false positives, the system administrators can extend the whitelist.

An IP address may contact a large number of community members either because the community is actually targetted or if the attacker is targetting all or most of the Internet (e.g., broad port scan). The system administrators may want to react differently to these alternative scenarios. Therefore, for each IP address that has contacted a community member, our system keeps track of how many times it has communicated with IP addresses outside our communities of interest.

B. *Input data*

Our community-based alerting service uses netflow as its input data source (although other types of information could be utilized as well). Netflow is a standard data format collected and exported by most networking equipment, in particular, network routers. It provides summary information about each network communication passing through the network equipment. Specifically, a network flow is defined as an unidirectional sequence of packets that share source and destination IP addresses, source and destination port numbers, and protocol (e.g., TCP or UDP). Each netflow record carries information about a network flow including the timestamp of the first packet received, duration, total number of packets and bytes, input and output interfaces, IP address of the next hop, source and destination IP masks, and cumulative TCP flags in the case of TCP flows. Note, however, that the netflow record does not contain any information about the contents of the communication between the source and destination IP addresses.

The community-based alerting service requires access to netflow to/from each of the organizations in the community. Such data can be collected by each of the organizations in the community at their edge routers and then collected at a central location for processing. Alternatively, it can be provided by an ISP that serves a number of the organizations in the community. Note that the netflow data may be sampled (to reduce the volume of the data) and the actual IP addresses of the computers within each organization can be obfuscated prior to the analysis (e.g., all IP addresses belonging to an organization can be collapsed into one address) if desired.

Given the collected netflows and the IP address ranges belonging to each member organization in the community, our alerting service analyses the data (either real time or in daily or hourly batches) and generates alerts to the system administrators. The analysis algorithm is described in Section III. A whitelist can be used to eliminate any legitimate communication destinations from consideration (e.g., search engines, CDNs, banking, on-line retailers, etc).

III. Implementation

A. *Architecture*

The architecture of our system is presented in Figure 3. We use three different types of processing components that do not share any state and are executed as individual processes: the parse, the filter, and the graph components. Each component can be replicated and executed by any number of processes (e.g., L, M, $N1$, and $N2$ in the figure). Every process of every component has a unique id (from 0 to the number of processes for the component-1) that is used for message routing. Since the parse component is connected to the filter component, each parse process is connected to each filter process. The same is true for the filter and graph components. Note that the system supports multiple different kinds of graph components in one system configuration as illustrated by *Graph 1* and *Graph 2* in the figure. Different graph components can be used to realize different alerting conditions as we will describe below.

The communication between components is based on event messages that are sent via TCP-channels. A message consists of a key and a body that are defined by the pair of interacting components (e.g., parse and filter, or filter and graph) and may contain any information desired by these components. For the key, a hash function h must be available that maps the contents of a key into an unsigned integer, which is used to route the event message to the right receiving process. For example, if a parse process is connected to 2 filter processes (i.e., $M = 2$), the receiving filter process is chosen by calculating the modulo of the hash of the key and 2. Thus, in this particular example, all keys with even hashes would be routed to the first and all keys with odd hashes to the second filter process.

The internal state maintained by each component is partitioned by the same key, making it possible to distribute their processing load onto multiple cores efficiently.

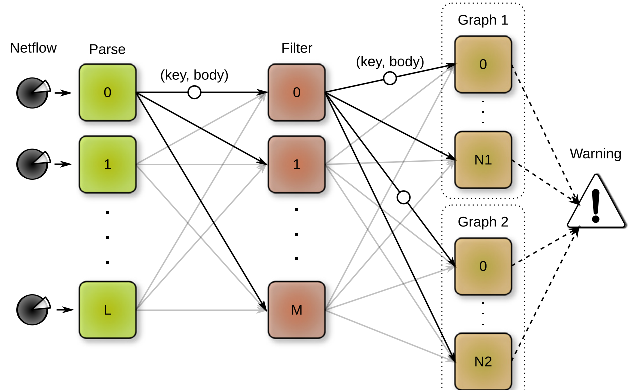


Fig. 3: Data processing architecture

Each network flow is processed as follows. First, the netflow data is read from a local storage device (it could also be received in real time from a router). The parse component transforms the IP addresses from their original string representation (i.e., “AAA.BBB.CCC.DDD”) into an integer representing the

IP address[3] and constructs a message with 5 fields: sourceIP, source-port, destinationIP, destination-port, and transferred-bytes. The parse component sends this message to the filter component. It uses the sourceIP field of the message as the key. The filter component either forwards (using the same key) or discards the received message. This decision is based on various factors, like used ports and source and destination IPs. If the message is forwarded, it is forwarded to one process of every graph component (e.g., *Graph 1* and *Graph 2*). Finally, the graph components construct a *community graph* for each source IP. The filtering and community graph construction are described in detail below.

B. Filtering

The filter is an essential part of our analysis and its role is to remove irrelevant flow records and to reduce the amount of data that needs to be processed by the graph component. For example, commonly used search, news, social media, and entertainment web sites are used so frequently that they would appear with almost every community. Furthermore, any traffic that does not involve at least one community member is not relevant for the analysis and is filtered out. Other filtering actions can be chosen based on data volume and perceived threat vectors. For example, HTTP-traffic may be filtered to reduce data volume, but at the risk of missing attacks that use HTTP (port 80).

1: Example Filter algorithm

input : (src-IP, src-port, dst-IP, dst-port, transferred-bytes)
output: The same as the input, if not filtered

```
//collapse IP addresses
src-IP, dst-IP = collapse(src-IP), collapse(dst-IP);
//filter IPs of commonly used web sites
if src-IP ∈ whitelist then
    return ∅;
end
//filter web-accesses to community-members
if dst-IP ∈ community then
    if src-IP ∉ community then
        if src-port = 80 then
            return ∅;
        end
    end
end
//only forward if one of the IPs is in the community
if dst-IP ∈ community OR src-IP ∈ community then
    return (src-IP, src-port, dst-IP, dst-port, transferred bytes);
end
```

Algorithm 1 shows an example filter component that filters connections based on their ports, and source and destination IP addresses. First, the algorithm collapses IP addresses for an organization into one address. If, for example, an organization has the IP range from `141.1.0.0` to `141.85.255.255` and either the src-IP or dst-IP are within this range, it is set to `141.1.0.0`. We then discard every connection from IP addresses that are contained in the whitelist. Second, accesses to a community member's web-server are filtered. Finally, we only forward the event message if at least one of the connection end-points is contained in the community.

[3]We will continue calling this identifier an IP address to enforce the one to one connection between these numerical IDs and the IP addresses.

C. Community Graph

We build a fixed size (K) Community of Interest (COI) graph for each IP address that is received by the graph component. Essentially, we use a windowed top-K algorithm, as described in [3]. However, there are two significant differences in our implementation compared to [3]. First, our window is not based on a fixed time interval, but rather on the observed connections. This has the benefit that the COIs of IP addresses with many connections will be updated more often than of those with very few. Second, we introduce several COI views ($\{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$) that use different methods to determine the weight of a connection. We can, for example, favor connections that transfer many bytes over those that only transfer a few by using the transferred bytes as the edge weight. Obviously, in this case we would not be able to detect attacks that transfer only a small set of data if these connections are dominated by large file transfers. Therefore, we define another view that uses the port numbers involved in security incidents to weight the edges (i.e., the more reported security incidents for a port, the larger the weight). Our system supports any number of such views running in parallel, as depicted in Figure 3 (with *Graph 1* implementing a different view than *Graph 2*).

Algorithm 2 shows how the COI is constructed in more detail. The algorithm uses two main data structures: a window that is used to collect recent data and a COI graph that stores the COI graph as seen from the beginning of the analysis run. We first add the received connection to the window. If more than 1000 connections have already been added, the window is merged with the COI graph. To this end, for each *IP* in the window, the weight of each edge is calculated, multiplied with a damping factor $1 - \theta$ and added to the weight in the COI, which is first multiplied with θ. Since $\theta = 0.85$, the influence of the new connections in the window is dampened. We also merge the port-mapping per destination-IP. It maps the source-port to the destination-port and a counter, counting how often this port-combination was used. Thereafter, the weights of all contacts in the COI that have not been observed during the current window are decayed by multiplying them with θ. To keep the COI at a maximum size of K, we remove the weakest links until the size of the COI is equal to K. Finally, the window and the counter are reset.

D. Generating Alarms

We showed above how the COI graph is constructed. Here, we provide two complementary algorithms to detect suspicious IP addresses.

The first, shown in Algorithm 3, is used to pre-filter all IP addresses that belong to a community. However, if a computer inside the community is compromised, we still want it to be checked further. To this end, we iterate over all connections in

```
2: Example Community graph construction
input  : (src-IP, src-port, dst-IP, dst-port, transferred-bytes), s =
         State[src-IP], F
output: None
//Save connection in window
s.window[dst-IP].transferred_bytes += transferred-bytes;
s.window[dst-IP].port_map[src-port][dst-port]++;
s.counter++;
//Merge window into topK after 1000 events
if s.counter > 1000 then
    foreach IP ∈ s.window do
        //θ has a value of 0.85 in our analysis.
        s.topk[IP].weight = 1 - θ * V(s.window[IP])
                            + θ * s.topk[IP].weight;
        //Merge the window's port map with the top-k's
        foreach {source-port, dest-port} ∈
        s.window[IP].port_map do
            s.topk[IP].port_map[source-port][dest-port] +=
            s.window[IP].port_map[source-port][dest-port];
        end
    end
    //Decay weight of old connections
    foreach IP ∉ s.window do
        s.topk[IP].weight = θ * s.topk[IP].weight;
    end
    //Remove the weakest links
    while size(s.topk) > K do
        remove_weakest_link_from(s.topk);
    end
    s.window = ∅;
    s.counter = 0;
end
```

the IP's top-K and check each pair of ports. The pairs of ports, considered suspicious, are specified using a configuration file.

We call Algorithm 4 for all IP addresses returned by Algorithm 3. It assures that (1) only those IP addresses that connected to at least `min_cnt` members of the community will be reported and (2) that the connections to the community make at least `min_part` percent of all the connections of the current IP address.

The detection algorithm can be run either for all IP addresses at once or individually for each IP address. Therefore, it is possible to provide different detection latencies. For example, to detect a suspicious IP address the earliest possible, the algorithm must be executed as soon as a message is received for its source-IP's top-K. If this is not necessary, the algorithm can be run for all top-Ks in one graph process at any desired interval.

The generated alarms can be emailed to the system administrators in the affected organizations or posted on a security dashboard. The reports contain the complete top-K for each suspicious IP address, including the port mappings.

```
3: Suspicious IP detection (1)
input  : IP, community, s = State[IP]
output: IP, if suspicious; ∅, if not
//blacklisted IPs are always suspicious
if IP ∈ blacklist then
    return IP;
end
//check if IP is in the community
if IP ∈ community then
    //iterate over all of IP's connections
    foreach conn ∈ s.topk do
        //iterate over all ports of one connection
        foreach p ∈ s.topk[conn].port_map do
            //check if src_port and dst_port are suspicious
            if is_suspicious(src_port, dst_port) then
                return IP;
            end
        end
    end
    //no strange ports -¿ skip
    return ∅;
end
//not in community -¿ check
return IP;
```

```
4: Suspicious IP detection (2)
input  : IP, community, min_cnt, min_part, s = State[IP]
output: Alarm
//check if top-K connections of this IP are in the community
  often enough
cnt = count_community(community, s.topk);
part = cnt / size(s.topk);
if IP ∉ blacklist then
    if cnt ≤ min_cnt OR part ≤ min_part then
        return false;
    end
end
return true;
```

IV. Evaluation

A. Input data and general setup

We currently run the experiment on a per-day basis. This means we fetch the netflow entries of the last 24 hours and run our analysis. We do not carry any state from one daily run to the next. In principle, we could leave the system running continuously or checkpoint the graph component and re-initiate its state on the next day. However, we found it useful to start with a clean system every day since this makes it easier to reason about the impact of changes in the community and white lists.

Moreover, we introduced the concept of different views in June 2011. Since then, we use three different views: one that weighs the bytes transferred, another that weighs the number of connections made, and the last one that weighs the security risk for the ports used (as described in Section III). For any measurements that were conducted before this date, we only used the view based on the bytes transferred.

Our input data-set is heavily sampled netflow from an ISP. In the first step, we remove all unimportant fields, leaving only the source-IP, destination-IP, source-port, destination-port, and the number of transferred bytes. This sums up to roughly

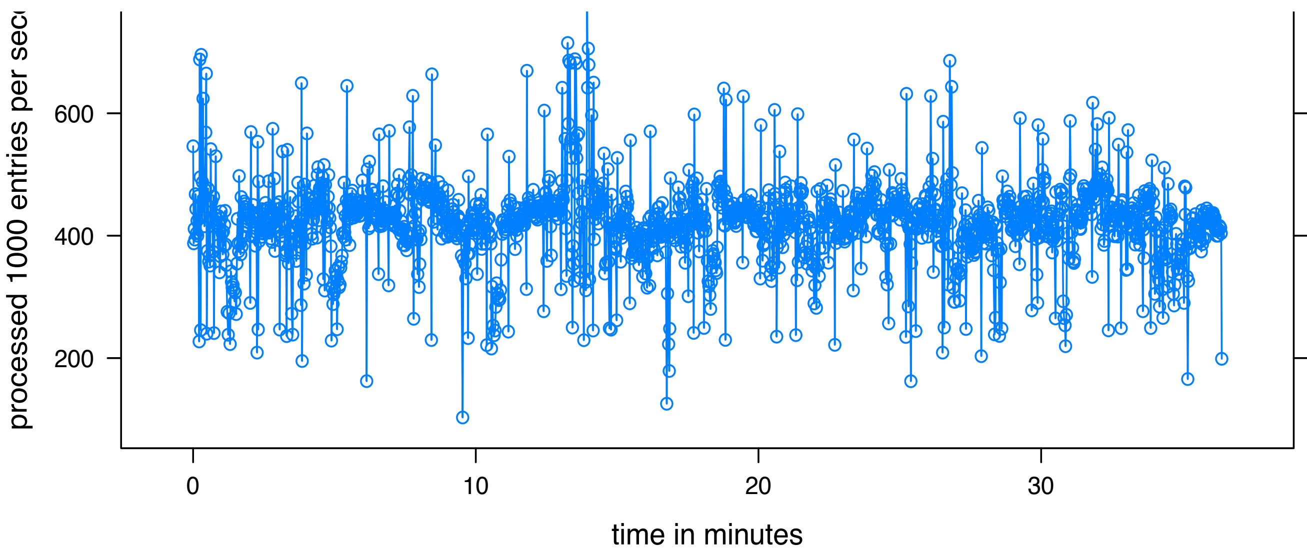


Fig. 4: Processed netflow-entries per second over the complete analysis execution

50GB of processed netflow per day.

The community lists define a community with the IP address ranges of all its members and each community is stored in a separate file (the white list is simply a "special" community). For example, if we wanted to add "TU-Dresden" to a "universities community" we would add the following line into the corresponding file:

```
141.1.0.0 - 141.85.255.255 TU.DRESDEN.DE
```

If a company or institution has more than one IP address range assigned, we can simply add each range as a separate entry. Moreover, an entry in one community is allowed to be a member in other communities as well.

B. Performance

We implemented the parse, filter, and graph components on top of StreamMine [12], a highly scalable stream processing system. While StreamMine supports scaling to hundreds of physical machines, a scalability and performance evaluation involving multiple machines is out of the scope of this paper. Therefore, we only used a single machine with 24GB of RAM and 16 processing cores for the analysis. For the top-K algorithm we used a value of 100 for K.

Figure 4 shows the read-throughput of the parse component of one such run in which we processed one day of netflow data (using only one view). The measurement was taken every second throughout the whole run. The parse component can read around 400,000 netflow entries per second with this single machine. Each entry is converted into a message and sent to the filter component. The filter component discards a large fraction of these messages and only send around one in a hundred of the incoming messages to the graph component. Naturally, the read throughput varies over time, since the amount of processing that needs to be done in the system depends heavily on the content of the input data. However, it is important to note that the mean throughput stays constant, i.e., the system performance does not decline with time as more graphs are added.

In the experiments reported in this paper, the filter component uses 13 of the available cores, since it has to filter the 400,000 netflow entries arriving every second. The graph component uses only one core since the amount of data it has to process is only a fraction of the data the filter receives. Note that even if one would assign more processing resources (i.e., cores) to the graph component, it would still be impossible to process unfiltered traffic (i.e., system without the filter component)— the system would simply run out of memory. The parse component uses the remaining two cores for reading the input files and parsing their contents.

To avoid queuing, StreamMine uses the TCP back-pressure mechanism on the network-connections. Hence, if a message cannot be processed by the filter component because all its threads are already busy processing other messages, the parse component will eventually stop sending new messages (the TCP send blocks if messages are not read fast enough on the other side). This will eventually lead to the parse component not reading any new netflow entries, because all its threads are blocked trying to send messages.

Figure 5 shows the size of the daily alarm report (= number of suspicious IPs communicating with the community) and community sizes (approximately the number of member organizations) over time for several months. The size of the

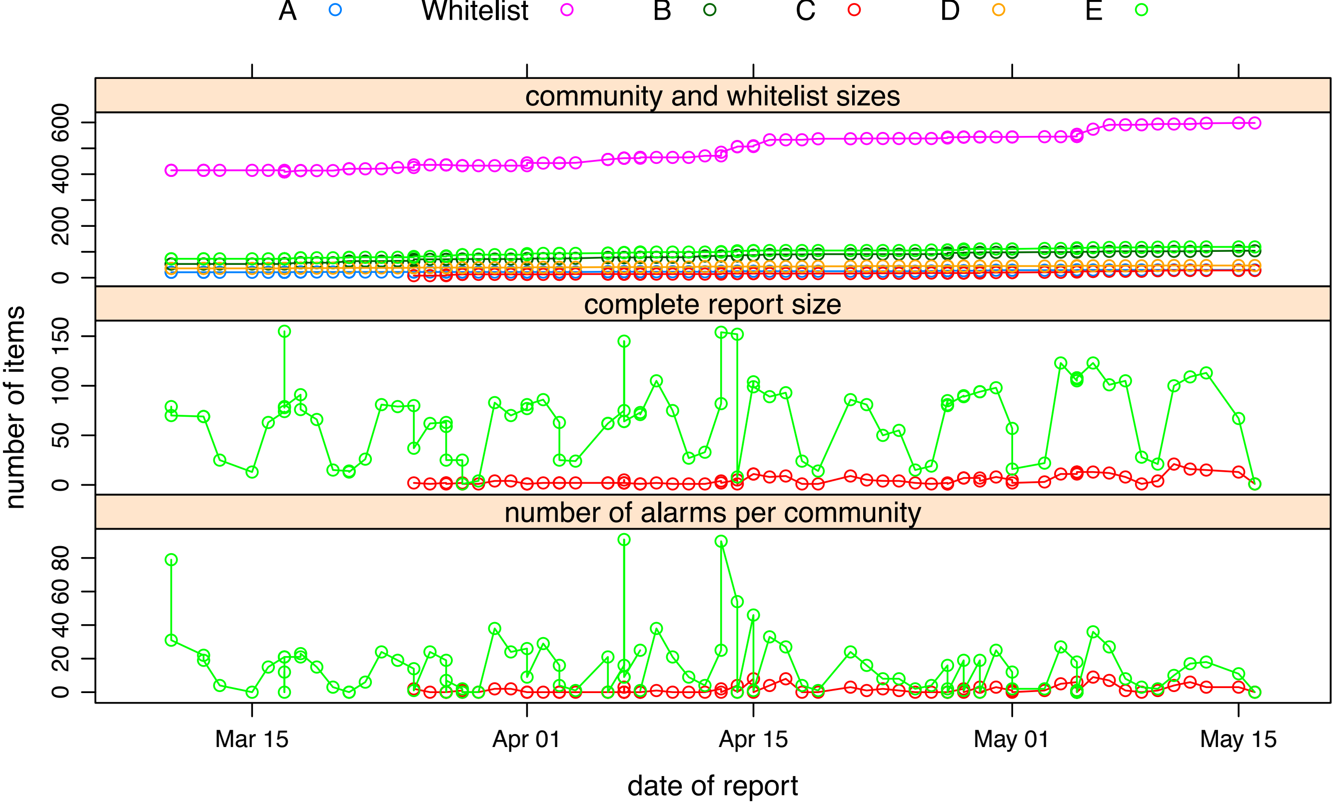


Fig. 5: Community and alarm sizes over time

alarm report is subject to a weekly pattern with larger sizes for weekdays (for alarm reports produced from Tuesday to Saturday) and smaller for weekend traffic. The community lists and the white list were updated manually on a daily basis. Given a fixed community, the community list would typically stay relatively fixed but in our case we occasionally identified additional community members. For the alarm reports, we only plot the report sizes for communities E and C. We did not generate reports for the other communities because (1) we found E and C to be the most interesting ones and (2) because of time-constraints as we need to scan the reports manually for attacks and new members of the community or white lists. It is natural that the reports, especially initially, contain a number of false positives. Some of them will be new community members that have to be added to the community list, while others are companies and organizations that can be added to the white list. The white list is used to filter out trusted traffic, i.e., from well known search engines, entertainment web sites, social media, popular CDNs, banking, government services, etc.

In an actual usage of the system, the system administrators analyzing the alarm reports would also add other known "good" IP addresses to the white list to prevent them from being reported daily. Lacking such domain knowledge, our experiments used the white list conservatively. The bottom plot approximates the size of the daily alarm report under real usage scenario where suspected IP addresses are processed daily and either added to the white list or the suspect communication is stopped (e.g., clean up infected machine, add firewall rules). This alarm size is approximated simply by only listing the IP addresses that have not been reported before.

V. Case Studies

While we do not typically know the ground truth, we have observed a number of suspicious cases in our analysis. In this section, we outline some of these examples.

A. *Case 1*

Table I shows an anonymized part of the report, generated for the netflow on May 13th, 2011. The report was obtained using the view based on the number of bytes transferred. It depicts the anonymized source-IP address (X.Y.Z.W) and the communities it was connected to, which ports were used (to help identify the application or service used), and a measure of the frequency of communication—the "Occurence" field indicates how often this connection was observed in the COI. In the actual report, the IP address and the exact community member are visible, of course.

In the next step, we usually use the *whois* service, to determine to whom the IP address belongs. This way, we may also find new members of the community by looking up the company names, displayed in the *whois* information. For this

IP Address	Src Port	Community	Dst port	Occurrences
X.Y.Z.W	6000	E	1433	2
X.Y.Z.W	6000	E	1433	2
X.Y.Z.W	6000	E	1433	1
X.Y.Z.W	6000	E,C	1433	1
X.Y.Z.W	6000	B	1433	1
X.Y.Z.W	6000	E,C	1433	1

TABLE I: Anonymized report-snippet (port-mapping) from May 13th, 2011

particular example, the only information we could get, was that it belongs to an Asian ISP. Since the IP address likely does not belong to a company that the community members would typically collaborate with, we have a closer look at the ports being used. We assume that the lower port number (1433) belongs to the server and the higher port-number to the client (6000). Figures 6 and 7 show the output of the "SANS Internet Storm Center" web-site[4] related to port 1433. The web-site shows the services that usually run on these ports—in this example, "Microsoft-SQL-Server". The SANS reports indicate many potential vulnerabilities, which may be used, for example, to steal data.

Unfortunately, this is usually everything we are able to derive from the netflow alone. While we consider this to be a potential attack, final certainty could only be provided by the system administrators of the individual companies, given they have deeper knowledge about legitimate communication connections of each organization and access to lower-level logs on the targeted machines.

B. Case 2

Table II shows a summary of the COI of another anonymized IP address for August 8th, 2011. It shows the IP address, each community and two numbers. The report was generated using the view based on the security risk of used ports. The first number is simply a count of how many members of the current community had an entry in the COI of this IP address. The second number shows how often the IP address connected to other IP addresses that are in none of the communities. We stated in Section II that this number is a good indicator of the severity and specificity of an attack. Here, it is relatively low, which leads to the assumption that the connections were not driven by a brute-force or port-scan-like technique.

To verify this intuition, Table III shows the used ports for each community member individually. In contrast to the previous example, the source port is not constant anymore but seems to be chosen randomly. The destination port, however, is constant 445. Port 445 is usually used by "Win2k+ Server Message Block". Note that every connection only appeared once in the netflow. This either means there was in fact just one connection being used or the attempt to connect failed.

In the next step, we use again the *whois* service, to determine that the IP address belongs to an European ISP. However,

[4]http://isc.sans.org

IP Address	Community	# in Top-K	# outside Community
X.Y.Z.W	A	0	42
X.Y.Z.W	B	0	42
X.Y.Z.W	C	1	42
X.Y.Z.W	D	0	42
X.Y.Z.W	E	1	42
X.Y.Z.W	F	6	42

TABLE II: Anonymized report-overview-snippet from August 8th, 2011. The last two columns contain the following numbers: (1) Number of members of the current community which had an entry in the COI of the current IP address and (2) number of connections to non-community members after the first connection to a community-member.

IP Address	Src Port	Community	Dst port	Occurrences
X.Y.Z.W	4798	F	445	1
X.Y.Z.W	1238	F	445	1
X.Y.Z.W	1256	F	445	1
X.Y.Z.W	1682	F	445	1
X.Y.Z.W	3143	C,E,F	445	1
X.Y.Z.W	4243	F	445	1

TABLE III: Anonymized report-snippet from August 8th, 2011

it is not clear if this address belongs to a community member. An attempt to *ping* the address did not succeed. A query to "SANS Internet Storm Center" (Figure 8) shows a long list of reports about worms using this port with the famous "Conficker" being one of them.

As with the previous example, we cannot determine if this case is a true attack. To this end, we would need the help of the system administrators of the various community members who have access to the log-files of the corresponding machines. However, there are two interesting points concerning this IP address. First, there are only a total of 69 entries in the netflow, where this address is the source of communication. Second, all connections transfer only a very small amount of data—around 60 bytes each. Even in total, this only sums up to several kilo bytes. Therefore, this address only appears in the ports view and not in the other views that consider either the number of bytes or connections. Hence, an administrator would need to set the detection threshold very low to see an alarm concerning this address.

C. Case 3

In contrast to the previous two cases, this case is not an attack. It occurred in all views and if one only looks at the report (an excerpt is shown in Table IV), it is not immediately clear what service is being used since the address seems to be using random ports on both ends of the communication. The query to *whois* does also not reveal any useful information, except that the address belongs to a US ISP.

However, looking at the connections with IP addresses outside of the communities provides a hint that this is not targeted against any of our specified communities as shown in

User Comment

Submitted By	Date
Comment	
Marcus H. Sachs, SANS Institute	2003-10-10 00:50:59
SANS Top-20 Entry: W2 Microsoft SQL Server (MSSQL) http://isc.sans.org/top20.html#w2 The Microsoft SQL Server (MSSQL) contains several serious vulnerabilities that allow remote attackers to obtain sensitive information, alter database content, compromise SQL servers, and, in some configurations, compromise server hosts. MSSQL vulnerabilities are well-publicized and actively under attack. Two recent MSSQL worms in May 2002 and January 2003 exploited several known MSSQL flaws. Hosts compromised by these worms generate a damaging level of network traffic when they scan for other vulnerable hosts.	
Johannes Ullrich	2002-10-10 17:21:35
Port 1433 is used by Microsoft SQL Server. SQLSnake is one worm taking advantage of SQL Server installs without password. As SQL Server is able to run batch files and command line programs, it can be used to download and install malware. Basic Protection: Use good passwords for all SQL Server accounts.	

Fig. 6: Screenshot of "`http://isc.sans.org/port.html?port=1433`" from September 8th, 2011

CVE Links

CVE #	Description
CVE-1999-287	"Vulnerability in the Wguest CGI program."
CVE-2000-1081	"The xp_displayparamstmt function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1082	"The xp_enumresultset function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1083	"The xp_showcolv function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1084	"The xp_updatecolvbm function in SQL Server and Microsoft SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1085	"The xp_peekqueue function in Microsoft SQL Server 2000 and SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1086	"The xp_printstatements function in Microsoft SQL Server 2000 and SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2000-1088	"The xp_SetSQLSecurity function in Microsoft SQL Server 2000 and SQL Server Desktop Engine (MSDE) does not properly restrict the length of a buffer before calling the srv_paraminfo function in the SQL Server API for Extended Stored Procedures (XP)
CVE-2001-542	"Buffer overflows in Microsoft SQL Server 7.0 and 2000 allow attackers with access to SQL Server to execute arbitrary code through the functions (1) raiserror
CVE-2002-642	"The registry key containing the SQL Server service account information in Microsoft SQL Server 2000

Fig. 7: Screenshot of "`http://isc.sans.org/port.html?port=1433`" from September 8th, 2011

Ports 445/TCP -- SMB DIRECT HOST" DMZ servers that are members of the internal domain." "Chapter 3 - Firewall Design - Infrastructure (Domain - SMB Direct Host) 445 TCP "additional protocol definitions that were created on the internal ISA Server firewall to all servers in the DMZ (IIS and DNS) to join and participate in the domain, and for the management agents installed on these servers to be able to forward information packets to the internal management servers." Table 2 New Protocol Definitions Protocol Definition Name - Direct Host (TCP) Internal Connection Port Number - 445 Initial Protocol - TCP Initial Direction - Inbound" "Active Directory Replication over Firewalls; Full dynamic RPC - Cons - Turns the firewall into "Swiss cheese" - Server message block (SMB) over IP (Microsoft-DS) 445/tcp, 445/udp. (Ask Us About... Security, March 2001 by Joel Scambray http://support.microsoft.com/default.aspx?scid=KB;en-us;289241&) Limited RPC - SMB over IP (Microsoft-DS) 445/tcp, 445/udp" "XCCC: Exchange 2000 Windows 2000 Connectivity Through Firewalls - Enable Windows 2000 Server-based computers to log on to the domain through the firewall by opening the following ports for inbound traffic: 445 (TCP) - Server message block (SMB) for Netlogon, LDAP conversion and distributed file system (Dfs) discovery."

Johannes Ullrich	2009-10-04 18:45:22

now also used by the "Lioten" worm/virus.

Bob A. Schelfhout Aubertijn	2009-10-04 18:45:22

As Johannes Ullrich stated wisely in his comment, 445 is also used by the Win2k / WinXP worm "Lioten" also known as "iraq_oil.exe". Since a couple of weeks or so firewall logs show a heightened incoming activity on Port 445, very likely due to this worm. FYI, following links can help you out when needed. http://www.f-secure.com/v-descs/lioten.shtml http://vil.nai.com/vil/content/v_99897.htm http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.lioten.html Stay happy, stay clean.

Deb Hale	2009-10-04 18:45:22

New Worm detected by Symantec on 06/07/03. Maybe what we are seeing the last couple of days. W32.Randex.B is a network-aware worm that will copy itself to the following paths: \Admin$\system32\msslut32.exe \c$\winnt\system32\msslut32.exe on computers with weak administrator passwords When W32.Randex.B is executed, it does the following: Caclulates a random IP address for a computer to infect. The worm will not infect computers with IP addresses in the following ranges: 10.0.0.0 -> 10.255.255.255.255 172.16.0.0 -> 172.16.255.255 192.168.0.0 -> 192.168.255.255 127.0.0.0 -> 127.255.255.255 240.0.0.0 -> 240.255.255.255 Attempts to authenticate itself to the aforementioned randomly-generated IP addresses using one of the following passwords: <blank> admin root 1 111 123 1234 123456 654321 !@#$ asdf asdfgh !@#$% !@#$%^ !@#$%^& !@#$%^&* server Copies itself to computers (with weak administrator passwords) as the following: \<authenticated IP>\Admin$\system32\msslut32.exe \<authenticated IP>\c$\winnt\system32\msslut32.exe Schedules a Network Job to run the worm: Adds the value: "superslut"="msslut32.exe" to the registry key: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run so that the worm runs when you start Windows.

DK*CERT	2009-10-04 18:45:22

The new "Deloader" worm also uses this port. http://www.f-secure.com/v-descs/deloader.shtml

anonx	2009-02-09 15:56:30

Its Conficker.B hammering the port at the moment. It operates in several modes (not at same time). One mode tries to get out to sites on web and the other tries to crack passwords on accounts (I think it starts by going through host file..)this results in account lockouts)- the 2 together form a very effective DDoS on corporate networks - causing major DNS/AD problems. Not sure if there is third mode which is just spreading itself (or whether the other 2 do that)- it sets scheduled jobs to rundll multiple infections at once. From my experince Oct MS patch doesn't always work. Tuesday's patch from MS and updated malicious software removal tool better. We have cured about 600 infected servers and PCs and still got some to go...

greyfairer	2008-12-11 01:08:28

Hmm, seems like a new variation has broken out: sources/day x 6 I guess a lot of people have been infected by the Gimmiv.A virus this weekend: http://blog.threatexpert.com/2008/10/gimmiva-exploits-zero-day-vulnerability.html

Luis	2006-01-07 00:30:59

We have some clients with malwares and process: adtech2006a Access to page: http://www.findthewebsiteyouneed.com/ Scans sequential ips (10/seg) using 445 port. Solutions: ad-aware se and windows update if necessary. Some clients with an anti-spyware not detected malware or malwares.

Bill Pipes	2005-06-22 02:40:54

We were hit hard with W32/Sdbot worm that's associated with the MS LSASS vulnerablity (ms04-011). We had some machines that weren't patched and

Fig. 8: Screenshot of "`http://isc.sans.org/port.html?port=445`" from September 8th, 2011

IP Address	Src Port	Community	Dst port	Occurrences
X.Y.Z.W	13397	B	38426	1
X.Y.Z.W	41748	F	41387	1
X.Y.Z.W	49534	C	23068	1
X.Y.Z.W	16249	C	22654	1
X.Y.Z.W	29167	C	43183	2
X.Y.Z.W	20	F	7205	4
...	...	...	...	...

TABLE IV: Anonymized report-snippet from August 8th, 2011

IP Address	Community	# in Top-K	# outside Community
X.Y.Z.W	A	0	14250
X.Y.Z.W	B	2	14250
X.Y.Z.W	C	1	14250
X.Y.Z.W	D	0	14250
X.Y.Z.W	E	0	14250
X.Y.Z.W	F	6	14250

TABLE V: Anonymized report-overview-snippet from August 8th, 2011. The last two columns contain the following numbers: (1) Number of members of the current community which had an entry in the COI of the current IP address and (2) number of connections to non-community members after the first connection to a community member.

Table V. Moreover, the use of port 20 (the last line in Table IV) gives a hint that at least some part of the communication involved anonymous ftp, which uses port 20 to initiate the connection but uses random ports thereafter. Finally, using an ftp-client (i.e., a web-browser) revealed indeed that this is simply an ftp server hosting software updates. As a result of this analysis, we added the address to the white list.

D. Building Communities

In real use of the system, the community members might be known a priori and even stay relatively fixed. However, in our case we built the community lists incrementally by identified new community members based on the COIs gener-

ated. Specifically, we assumed that members of a community exchange information with one another and often the data exchange is encrypted. Therefore, we focused on new IP addresses that used the https-port (443) for communication. However, a certain minimal set of known members is needed before reports can be generated. This set should be as large as possible for two reasons. First, the likelihood that an unknown address that belongs to the community (and thus, should be added) connects to one or more entries of a large set of members is higher than if the set contains only very few entries. Second, if the set is large, one can set the reporting threshold higher and reduce the amount of noise.

Building a community this way is a task that lasts for weeks, depending on how much communication is observed between the individual members and how large the community is initially. We start by adding the new community to the list of communities. With every subsequent report, we scan for new members and add them to the corresponding lists. This way, the community grows every day, and with it the likelihood of finding any missing members. The community stabilizes eventually with fewer and fewer new members per day.

VI. Related work

A number of tools and techniques have been developed to process and visualize netflow data(see [17] for a survey). Netflow processing tools include OSU flow-tools [16], SiLK [7], and Nfdump[5]. In addition to command line tools, numerous graphical user interfaces exist to visualize and query network activity, including NTOP[6], Nfsen [9], NfSight [1], VisFlowConnect [20], FlowScan [14], NetPY [2], FloVis [18], VIAssist [5], and NFlowVis [6]. While visualization tools allow the users to view the netflow data from different perspectives to locate suspicious activity, our approach analyzes the data and produces small number of meaningful alarms each day. Also, our focus on communities allows us to detect attacks and suspicious behavior that is focused on a potentially small community, but would not show significantly on a global scale.

Detection of similar communication behavior in multiple hosts has been used previously to raise suspicion that hosts with the correlated behavior may be members of the same botnet. For example, [21] uses netflow data to identify sets of suspicious hosts and then uses host level information (collected on each host by a local monitor) to confirm or reject the suspicions. However, detection of botnets is simplified by the fact that the bots typically act in unison (e.g., start spamming or DDoS attack against a target at the same time). Indeed, much of the work in this area (e.g., BotMiner [8]) specifically build detection mechanisms based on the assumptions of the communication behavior required for a botnet. Furthermore, to our knowledge, prior work is limited to detecting similar behavior within one organization.

The concept of using a community to help detect security events has been used in the past. For example, the Ensemble [15] system detects applications that have been hijacked by using the idea of a trusted community of users contributing system-call level local profiles of an application to a common merging engine. The merging engine generates a global profile that can be used to detect or prevent anomalies in application behavior at each end-host in real time. A similar concept of collaborative learning for security [13] is applied to automatically generate a patch to the problematic software without affecting application functionality. PeerPressure [19] automatically detects and troubleshoots misconfigurations by assuming that most users in the community have the correct configuration. Cooperative Bug Isolation [11] leverages the community to do statistical debugging based on the feedback data automatically generated by community users. Vigilante [4] apply the community concept for containment of Internet worms by community members running detection engines on their machines, where the detection engines distribute attack signatures to other community members when a machine is infected.

[5]http://nfdump.sourceforge.net

[6]http://www.ntop.org

VII. Conclusions

In this paper, we have presented a community-based analysis and alerting technique for detecting small-footprint attacks targeting communities of interest for attackers such as financial institutions, e-commerce web site, or the electricity generation and distribution infrastructure. By comparing communication behavior across the member organizations in the community, it is possible to detect suspect behavior that may fall below detection thresholds at individual member organizations. A white list can be used to avoid repeating false positives. We have implemented the analysis algorithm in a scaleable distributed architecture that can process large volumes of netflow data efficiently.

References

[1] R. Berthier, M. Cukier, M. Hiltunen, D. Kormann, G. Vesonder, and D. Sheleheda. Nfsight: netflow-based network awareness tool. In *Proceedings of the 24th USENIX LISA*, 2010.

[2] A. Cirneci, S. Boboc, C. Leordeanu, V. Cristea, and C. Estan. Netpy: Advanced Network Traffic Monitoring. In *Proc. Int Conf. on Intelligent Networking and Collaborative Systems (INCOS'09)*, pages 253–254, 2009.

[3] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. Communities of interest. In Frank Hoffmann, David Hand, Niall Adams, Douglas Fisher, and Gabriela Guimaraes, editors, *Advances in Intelligent Data Analysis*, volume 2189 of *Lecture Notes in Computer Science*, pages 105–114. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44816-0_11.

[4] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[5] A.D. D'Amico, J.R. Goodall, D.R. Tesone, and J.K. Kopylec. Visual discovery in computer network defense. *IEEE Computer Graphics and Applications*, 27(5):20–27, 2007.

[6] F. Fischer, F. Mansmann, D.A. Keim, S. Pietzko, and M. Waldvogel. Large-scale network monitoring for visual analysis of attacks. In *Proc. Workshop on Visualization for Computer Security (VizSEC)*, page 111. Springer, 2008.

[7] C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas. More NetFlow tools: For performance and security. In *Proc. 18th USENIX LISA*, pages 121–132, 2004.

[8] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the USENIX Security Conference*, 2008.

[9] P. Haag. Watch your Flows with NfSen and NFDUMP. In *50th RIPE Meeting*, 2005.

[10] V. Igure, S. Laughter, and R. Williams. Security issues in scada networks. *Computers & Security*, 25(7):498 – 506, 2006.

[11] Ben Liblit. *Cooperative bug isolation: winning thesis of the 2005 ACM doctoral dissertation competition*. Springer-Verlag, Berlin, Heidelberg, 2007.

[12] André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker de Brum, Stefan Weigert, Andrey Brito, and Christof Fetzer. Low-overhead fault tolerance for high-throughput data processing systems. In *ICDCS '11: Proceedings of the 2011 31st IEEE International Conference on Distributed Computing Systems*, page TBD, Los Alamitos, CA, USA, June 2011. IEEE Computer Society.

[13] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. Ernst, and M. Rinard. Self-defending software: Automatically patching security vulnerabilities. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[14] D. Plonka. Flowscan: A Network Traffic Flow Reporting and Visualization Tool. In *Proc. 14th USENIX LISA*, pages 305–318, 2000.

[15] F. Qian, Z. Qian, Z. M. Mao, and A. Prakash. Ensemble: Community-based anomaly detection for popular applications. *5th International ICST Conference on Security and Privacy in Communication Networks*, May 2009.

[16] S. Romig, M. Fullmer, and R. Luman. The OSU flow-tools package and CISCO NetFlow logs. In *Proc. 14th USENIX LISA*, pages 291–304, 2000.

[17] C. So-In. A Survey of Network Traffic Monitoring and Analysis Tools. Cse 576m computer system analysis project, Washington University in St. Louis, 2009.

[18] T. Taylor, D. Paterson, J. Glanfield, C. Gates, S. Brooks, and J. McHugh. FloVis: Flow Visualization System. In *Proc. Cybersecurity Applications and Technologies Conference for Homeland Security (CATCH)*, pages 186–198, 2009.

[19] H. Wang, J. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *In OSDI*, pages 245–258, 2004.

[20] W. Yurcik. VisFlowConnect-IP: a link-based visualization of Netflows for security monitoring. In *18th Annual FIRST Conf. on Computer Security Incident Handling*, 2006.

[21] Y. Zeng, X. Hu, and K. Shin. Detection of botnets using combined host- and network-level information. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

WCIS: A Prototype for Detecting Zero-Day Attacks in Web Server Requests

Melissa Danforth
Department of Computer and Electrical Engineering and Computer Science
California State University, Bakersfield
melissa@cs.csubak.edu or mdanforth@csub.edu

Abstract

This work presents the Web Classifying Immune System (WCIS) which is a prototype system to detect zero-day attacks against web servers by examining web server requests. WCIS is intended to work in conjunction with more traditional intrusion detection systems to detect new and emerging threats that are not detected by the traditional IDS database. WCIS is at its core an artificial immune system, but WCIS expands on the concept of artificial immune systems by adding a classifier for web server requests. This gives the system administrator more information about the nature of the detected threat which is not given by a traditional artificial immune system. This prototype system also seeks to improve the efficiency of an artificial immune system by employing back-end, batch processing so that WCIS can detect threats on higher capacity networks. This work shows that WCIS is able to achieve a high rate of accuracy at detecting and classifying attacks against web servers with very few false positives.

Tags: Research, Security, Web, Artificial Immune System

1 Introduction

Traditional intrusion detection systems (IDS) are very efficient at detecting known threats and even some emerging variants, but are not as effective at detecting zero-day attacks. Artificial immune systems (AIS) are appealing for detecting zero-day attacks because they are inspired by the adaptive concepts of biological immune systems. Biological immune systems are alluring to the computer security realm because they can innately adapt to new pathogens or variations on previously seen pathogens, something which even modern intrusion detection systems struggle to do. The primary goal of an artificial immune system is to apply these biological principles to the problem of distinguishing normal traffic or data from abnormal traffic or data, even if the abnormal traffic corresponds to a completely new attack.

This work presents a variation of the artificial immune system concept called Web Classifying Immune System (WCIS). WCIS is intended to work in concert with a traditional IDS, scanning the traffic that the IDS has labeled as normal to see if there is a zero-day attack, or even just a new, unknown variant of an existing attack, present in the traffic. As the name implies, WCIS focuses on attacks conveyed in web server requests. While the concepts can apply to other problem domains, this work focuses on web server requests as a "proof of concept".

There are limitations to the traditional AIS model that WCIS seeks to overcome. Most traditional artificial immune systems only provide this binary classification of traffic or data as "normal" or "attack". For many problem domains, particularly the problem domain of malicious web server requests, this simple classification is not sufficient. There are a variety of web server attacks ranging from simple information gathering via HEAD or OPTIONS requests to attacks that attempt to execute code on the web server. The administrative response to an attack will vary based on the type of attack. The prototype system presented in this work overcomes this limitation by adding classifications to a traditional AIS.

Since WCIS classifies the attacks as they are detected, this provides the web administrator with more information about the nature of the attack than a simple alert would provide. For example, an attack which has a directory traversal component would require different configuration changes than a CGI or PHP script with a buffer overflow. By providing classifications along with alerts, WCIS can help direct the administrative response to a zero-day attack more effectively. The administrators might not know the name of the attack, but if they know it's a buffer overflow on index.ida, that will allow them to focus their response far more than they could with an "attack detected" alert provided by traditional artificial

immune systems.

Another limitation of traditional artificial immune systems is the training of the immune system "antibodies", e.g. the sensors for detecting attacks. The traditional AIS model assumes a continual process of evolution occurring in real-time as it sees and classifies network traffic. Most evolutionary algorithms require extensive memory and CPU cycles to operate. This leads to two main issues using AISes when high-volume, real-time detection is desired: the sensors take a long time to train, during which they are not capable of accurately labeling traffic, and sensor refinement after initial training can cause a CPU and/or memory bottleneck that limits the volume of traffic that the sensors can process.

WCIS seeks to minimize these issues by separating the evolutionary processes from the detection process. The evolutionary processes, pre-deployment training and sensor refinement, occur "offline" on a back-end system. The detection process, monitoring the network traffic, occurs "online" in real-time on the network. The "offline" evolutionary processes produce a set of sensors, which essentially detect patterns in the traffic, that are deployed to monitor the network traffic in real-time. It should be noted that the "online" mode of WCIS is intended to work in conjunction with a traditional IDS by scanning the traffic which the traditional IDS has not alerted upon. WCIS however does not produce traditional IDS rules as those rules would be unable to gather the statistics at the sensor, classification population and overall population levels that are needed for sensor refinement.

In order to maintain one highly desirable feature of an AIS, the customization of the sensors for that particular network's traffic, WCIS uses a system profile to train the sensors in the pre-deployment phase. These profiles include a sampling of normal traffic for the network which will be used to train the AIS and a set of labeled attacks that will be used to "prime" the classifier. The prototype implementation of WCIS takes Apache logs as the source of these two datasets, which makes customization of the datasets very easy. One simply has to copy log entries over into the appropriate dataset file and rerun the pre-deployment phase of WCIS.

To enable "offline" sensor refinement, the "online" WCIS sensors record statistics about their detection and classification rates at the individual sensor, classification population and overall sensor population levels. This information can be sent to a back-end system, which will enable WCIS to run the sensor refinement process as a batch process on the back-end system while the live sensors keep detecting. Once the batch process is complete, the live sensors can be replaced with the newly refined ("next generation") sensors. The current prototype does not yet implement this aspect as the prototype could not be run on live network traffic due to policies and bureaucratic limitations about collecting data that may contain personal or confidential information at the university. However, it is already supported by the internal structure of the sensors and merely requires a live network (or isolated network) test environment to implement and fully test this feature.

In summary, WCIS is a variation of an artificial immune system that is intended to work in conjunction with a traditional intrusion detection system to detect attacks that the IDS cannot yet detect. WCIS seeks to overcome the usability limitations of traditional artificial immune systems by adding a classifier to provide more information about detected attacks. Additionally, WCIS seeks to optimize the scalability of the AIS concept by separating the evolutionary processes from the detection process. This allows the resource intensive aspects of an AIS to occur "offline" on a back-end system rather than on the detection system.

Section 2 provides an overview of artificial immune systems and the biological principles that inspired them. Related work in the area of artificial immune systems and classifiers is presented in Section 3. The methodology used to add classifications to an artificial immune system is described in Section 4. Section 5 describes how WCIS models web server requests. The results of running WCIS on sample datasets is presented in Section 6 and conclusions drawn from these results are given in Section 7. Finally, future avenues of research and improvement for WCIS are discussed in Section 8.

2 Artificial Immune Systems

An artificial immune system (AIS) is a type of anomaly-based intrusion detection system (IDS) inspired by the adaptive nature of the biological immune system. A biological immune system has to be responsive to new and unknown pathogens while also recalling previously defeated pathogens to prevent a recurrence of illness. While not 100% effective at this task (e.g. auto-immune disorders and other immune system malfunctions), the biological immune system is more adaptive to new pathogens and variants of known pathogens than the analogous anomaly-based IDSes.

Using biological methods to create a better IDS is the core concept behind artificial immune systems. The goal of an AIS is to distinguish normal traffic (called "self" data) from abnormal traffic (called "non-self" data). It does so by creating immune "sensors" as analogs to biological immune cells. These sensors use pattern matching functions to determine if data is "non-self". Several key features of a biological immune system that serve as inspiration for an AIS are affinity maturation, negative selection and peripheral tolerance. Other features of bi-

ological immune systems can also be incorporated, but these are far more weighty concepts that are beyond the scope of this paper.

Affinity relates to pattern matching. Each immune cell (antibody) has a set of proteins on its surface that form a three dimensional "lock" pattern which can match the "key" pattern of proteins on the surface of a pathogen (also called an antigen). Affinity measures how "tightly" the lock and key patterns fit together, with a higher affinity meaning a tighter bond between the antibody and pathogen exists. Affinity maturation is the process of refining an antibody's lock pattern until it can tightly bind to a specific pathogen. This allows the body to "memorize" specific pathogen patterns, e.g. learn a "signature" for that pathogen. This is the basis of immunizations in biological immune systems. For AISes, affinity maturation allows generic immune system sensors to develop "signatures" for novel attacks or new variants of old attacks. This is accomplished by training the sensors against attack data in the pre-deployment phase and by refining the sensors during deployment using an evolutionary technique, such as a genetic algorithm.

Negative selection is a process for creating new immune cells that do not react to the body's own proteins ("self"). Most of the artificial immune system works focus on this feature of biological immune systems. The immune cells are initially created with a random pattern of "lock" proteins. The cells are then tested against a random sampling of "self" proteins and structures. If the immune cell has too high of an affinity for "self", it is destroyed. For AISes, this means the immune sensors are initialized with random patterns and each sensor is tested against a sample of "normal" data. Those which react too strongly to normal data are removed and replaced with a new randomly generated and tested sensor. A negative selection phase can be used along with affinity maturation to be sure that the sensors do not start reacting to normal data while they are developing an affinity for attack data.

Since negative selection uses a random sampling of "self" proteins to test new immune cells, there is a possibility that cells which are reactive to self will survive negative selection. Auto-immune disorders are caused by such cells. In an AIS, such sensors would lead to false positives, where normal traffic is labeled as an attack. The immune system has some protection against this by using peripheral tolerance. Peripheral tolerance deactivates or destroys immune cells that are too reactive to self proteins. Not many AISes explore the use of peripheral tolerance in their systems since it is hard to detect false positives automatically. One technique might be to have a human verify each alert and deactivate any sensor which is noted to have an excessive number of false positives. In WCIS, the person can also modify the sensor's internal statistics to mark the sensor as "bad", which will prevent the sensor from being used to refine the sensors during the next sensor refinement phase. This essentially removes the sensor from the "genetic pool" used for sensor refinement.

3 Related Work

The research group of Stephanie Forrest at the University of New Mexico has produced several pioneering works in the field of artificial immune systems. Forrest, *et al.* [9] focused on distinguishing self from non-self and laid the foundations for the negative selection algorithm. Somayaji, Hofmeyer and Forrest [16] explored the application of these concepts to computer security. This work ultimately resulted in the production of the LYSIS [12, 13] immune system for TCP connections. LYSIS monitored the TCP/IP headers of SYN packets to detect abnormal traffic.

Williams, *et al.* [22] expanded LYSIS to monitor TCP, UDP and ICMP traffic. This system, called CDIS, also monitored all packets instead of just TCP SYN packets. Each AIS sensor in the system monitored a random subset of features from the packet headers. The pattern matching function used by CDIS used a mix of binary, discrete and real value features.

Gonzales, Dasgupta and Gomez [10] showed that the negative selection algorithm is very sensitive to the type of matching function used. Ultimately, one hopes that negative selection results in sensors with a wide coverage of the non-self space, as this represents potential attacks. But [10] showed that the algorithms of Forrest, *et al.* [1, 9, 12, 13, 16] and Farmer, *et al.* [8] resulted in restricted coverage of the non-self space. These algorithms work best with binary and discrete data. Of the algorithms tested, the real value matching function used by Gonzales, Dasgupta and Kozma [11] had the best coverage of the non-self space.

In Dasgupta, Yu and Majumdar [5], a multilevel immune learning algorithm was introduced, in part to overcome deficiencies in simple negative selection algorithms. This system used collaborations and interactions between various types of sensors, analogous to the various types of immune system cells in a biological immune system. By requiring collaborations between sensors to label data as non-self, the experiments showed the AIS achieved better results than simple negative selection.

The first version of WCIS that was published [2, 4] was built off the work of CDIS and LYSIS to monitor web server requests. As with CDIS, the sensors monitored a random subset of features and the pattern matching function used a mix of binary, discrete and real value features. The system also incorporated basic collaboration between sensors to reduce the false positive rate.

Table 1: The classification scheme used for the web server attacks.

Class	Instances	Description
info	5	Gathers information about server (read only)
traversal	37	Directory traversal attempt (read only)
sql	4	SQL injection attack
buffer	7	Buffer overflow attack
script	86	Cause a script to do something malicious (execute)
xss	40	Cross site scripting

This first version of WCIS was simply an AIS for web server requests and included none of the enhancements that this work covers. The concept of adding a classifier to WCIS was explored in [3], but the classifier used in that version was prone to overfitting and poor classifications and was deemed unsuitable. Neither previous version separated the evolutionary processes from the detection process.

Watkins, Timmis and Boggess [17, 18, 19, 20, 21] proposed an artificial immune recognition system for for supervised learning and reinforcement learning. The proposed AIS functioned as a classifier. As with [5], it modeled a variety of immune cells working in collaboration to classify data. It required the features be represented as a vector of real value ranges and used vector mathematics to calculate affinity and distance between cells. A variation on k-nearest neighbors was used to calculate the class of unknown data once the cells had been trained. While this method worked well on datasets that can be modeled as a feature vector, its mathematical approach limits its application to other feature sets that cannot be easily modeled as a vector.

4 Methodology for the Classifying AIS

Previously [2, 3, 4], WCIS was defined as an AIS for web server attacks and a rudimentary, but poor, classifier was implemented. The scheme for fingerprinting web server requests, detailed in Section 5, was developed in those works. The classifier developed in [3] was prone to overfitting and misclassification. A better classifier was developed, which is the focus of this section. The simple classification scheme given in Table 1 was preserved from [3] however, as the classification scheme was not the issue with the previous classifier.

The classification scheme in Table 1 was developed based on several common groups of web server request attacks that can be found encoded in URIs. The "info" classification covers various information gathering attacks that do not alter the server. Likewise, the "traversal" category solely covers the attacks which utilize directory traversal, but do not attempt to execute anything on the web server, such as attempting to read /etc/passwd. If the traversal tries to execute a program, it is instead labeled "script". The "script" class also covers other attempts to maliciously execute a program or script on the web server. The "sql" class covers SQL injection attacks. The "buffer" class covers buffer overflow attacks, which may also result in commands being executed. Finally, the "xss" class covers cross site scripting attacks. Table 2 lists some examples for each class except buffer overflow attacks as those examples were too long to easily fit into the table.

The classification training occurs during the pre-deployment stage where the field of potential sensors is trained against a system profile. The system profile consists of a normal dataset (Apache log entries from non-malicious web requests) and an attack dataset (Apache log entries from actual attacks on a web server). Each attack in the attack dataset was hand inspected and labeled with a classification. One main issue faced while developing the attack dataset was obtaining sufficient examples of each classification of attack. Attack examples were gleaned from Bugtraq [15], live Apache web servers and an un-networked machine where selected attacks were run against a local web server. As seen from Table 1, most of the examples fell into the category of traversal, script or xss. To prevent the sensors from becoming biased towards those classes, each sensor tracks the percentage of the class that it is able to detect rather than a raw count.

To add classification to WCIS, each sensor not only tracks the percentage of each category it reacted to during pre-deployment training, it also has a desired category for which it should develop affinity. Previously in [3], WCIS did not have this second feature and it was discovered that the population of sensors optimized for the "script" and "traversal" classes. To prevent this from happening, the sensors were divided into groups and each group was tasked with optimizing affinity for a particular classification. This is a niching algorithm, which is intended to develop "specialists" for all classification labels.

To optimize affinity, the sensors must be trained and matured. This is accomplished with a typical artificial immune system lifecycle conducted during the pre-deployment and sensor refinement phases. The lifecycle is an iterative process which repeatedly applies the affinity maturation steps. This results in a set of trained sensors that have higher affinity towards attacks than the initial sensors. The steps for the lifecycle are detailed in the following subsections.

The primary difference between the pre-deployment

Table 2: A sample of requests in the attack dataset.

Class	URL
info	GET x HTTP/1.0
traversal	GET ../../../boot.ini HTTP/1.0
traversal	GET %2E%2E/%2E%2E/%2E%2E/%2E%2E/%2E%2E/winnt/win.ini HTTP/1.0
sql	GET /scripts/test.asp?var=foo‘;EXEC master.dbo.xp_cmdshell’cmd.exe’ HTTP/1.0
script	GET /scripts/..%%35%63../winnt/system32/cmd.exe?/c+cmd.exe HTTP/1.0
script	GET /ans.pl?p=../../../../bin/command%20argument|&blah HTTP /1.1
xss	GET /<script>alert(’Vulnerable’)</script> HTTP/1.1
xss	GET /javascript:void%20window.open(HTTP/1.0

Table 3: A sample of requests in the normal dataset.

GET /00master/hqafgate.gif HTTP/1.0
GET /Copy%20of%2010.gif HTTP/1.0
GET /faq/web/viewfaq.php3 HTTP/1.0
GET /forums/newmsg.php?fid=2&pid=30 HTTP/1.1
GET /index.html?browsePage=commands.html HTTP/1.1
GET /index.html?browsePage=kb/item_detail.php&id=19 HTTP/1.1
GET /index.html?secure=1&PHPSESSID=db80c486ee8cef8090a532b93619cd7a HTTP/1.1
GET /%7E930www/Images/front_y2k_logo02.jpg HTTP/1.0
GET /ADTracker.asp?linkid=AHCX030&linktype=Room&RID=8 HTTP/1.0
GET /CGI-BIN/centralad/getimage.exe/19980714243?GROUP=default_buttons HTTP/1.0

and sensor refinement phases is the source of the statistics used for training. In the pre-deployment phase, training statistics come purely from the sensor's reaction to the system profile datasets. In the sensor refinement phase, statistics come from the sensor's reaction to live traffic, with negative selection against the normal dataset also conducted to prevent sensors from reacting to normal traffic.

Before going into the details of the pre-deployment phase, some key terminology should be reviewed. The sensor **population size** is the number of unique sensors being processed. Each individual sensor within the population has its own data structure to store its pattern, classification label and statistics. Patterns may be repeated in multiple individual sensors within the population. This is called a loss of diversity or **overfitting** which essentially leads to redundancy (e.g. multiple sensors have the same "signature"). The sensor **lifecycle** is the process of creating, refining and perhaps destroying individual sensors within the population. Throughout the lifecycle, the population size remains constant. Every destroyed sensor is replaced with exactly one sensor. The sensors that exist in each iteration through the lifecycle process are called a **generation** of the population. Each new generation is generated by the affinity maturation process, which uses a genetic algorithm to refine the sensor population *as a whole*. The sensor's **chromosome** is a method to represent the sensor's pattern by using data structures that can be manipulated by a genetic algorithm. The chromosome contains all possible features that a pattern in WCIS may use (see Section 5 for a description of the features), the current values for each feature and a flag to indicate if the sensor is using that feature in its pattern (e.g. if the feature is **expressed** in that particular sensor). The **fitness** of a sensor is determined by its statistics and is used to gauge its accuracy at detecting attacks in its classification label. The most fit sensors contribute more "genetic information" to the next generation than the less fit sensors.

4.1 Lifecycle

In pre-deployment training, a normal dataset, samples of which can be seen in Table 3, and the labeled attack dataset are given as input to the lifecycle function. The pre-deployment lifecycle begins by randomly generating a population of sensors for each classification group. The random generation process selects a subset of features for each sensor's matching pattern and randomly assigns values to those features. In the sensor refinement phase, the lifecycle function would instead begin with copies of the existing sensors and any sensors which have been deactivated by the system administrator (peripheral tolerance) will be discarded and replaced by a random sensor.

For both phases, the iterative affinity maturation process is then entered, which refines the sensors over a series of generations. It is important to note that affinity maturation occurs within each population for a classi-

fication label, not across all classification label populations. The goal of affinity maturation is to produce sensors which specialize in detecting attacks for that particular classification label, so each population is kept distinct.

4.2 Negative Selection Phase

The affinity maturation process begins with negative selection. The population of sensors is compared to the normal dataset. Any sensor that has too strong of an affinity to requests in the normal dataset is discarded and replaced with a random sensor. The replacement is likewise tested against the normal dataset and is not allowed to replace the discarded sensor until its random feature set (e.g. pattern) does not have strong affinity towards the normal dataset. The exact level of affinity towards the normal dataset that is tolerated in this phase is tunable in WCIS.

4.3 Training Phase

After negative selection, the sensors enter two phases of training. During the first phase of training, the sensors are compared to all of the attack requests and a random subset of normal requests. If a sensor has affinity to an attack, it records the classification of that attack. At the end of the first phase, each sensor will know the percentage of attacks in each category it can detect. It then sees which classification it is best at detecting and marks that classification as its class. The sensor may mark itself as a different classification than what its group is supposed to be optimizing for. This simply means the sensor is not as good at detecting the desired classification as it is at detecting a different classification.

During the second phase of training, the sensors make a second pass over the attack dataset. For each attack, the sensors which can detect it vote on the classification of the attack. The accuracy of each group of sensors at detecting its desired classification is recorded. This second phase is purely for computing the accuracy statistics and does not affect the affinity maturation process. The accuracy of the sensors during experimental testing is given in Section 6.

4.4 Genetic Algorithm Phase

After training, the sensors move on to the genetic algorithm phase. This phase first "breeds" the sensors to create the next generation of sensors and then mutates the next generation. The breeding phase uses a single-objective genetic algorithm which optimizes for a single fitness metric (multi-objective algorithms allow optimization for multiple fitness metrics). The fitness of each sensor for this phase is its ability to classify the attacks in the desired classification for its population. For example, if a "script" sensor can detect 70 of the 86 script attacks, it would have a fitness of 0.814 even if it could also detect 100% of the "traversal" attacks. A secondary fitness value is also computed for each sensor but is not directly used by the genetic algorithm. This fitness value measures how well the sensor can detect attacks without excessive false positives. The secondary fitness ranges in value from -2 (all of its alerts are on normal requests instead of attack requests) to +2 (all of its alerts are attack requests).

Rank selection with elitism using the primary fitness value is used to select the "parent" sensors. Rank selection chooses the most fit sensors to be parent sensors. Elitism allows a percentage of highly fit parent sensors to survive into the next generation. The exact percentage is tunable in WCIS. Once two parent sensors are selected, single point crossover on the parents' chromosomes is used to create the chromosomes for the "children" sensors. The chromosome is the complete feature set, a subset of which will be expressed in each parent. The expressed feature set for each child sensor is the intersection of the expressed feature sets of the parent sensors. Additionally, a feature that only one parent expresses will be randomly expressed in the child. Even if the feature is not expressed, the child will still inherit the values for that feature from the parent. It just will not be used by the child to match against requests. But this preserves the genetic information in a dormant state in case future offspring randomly choose to express that feature. Finally, if a child exits this expressed feature selection phase with less than two features expressed, it randomly chooses features to add to its expressed feature set until the set size is two.

Besides the children sensors created by crossover, randomly selected parent sensors are also be chosen as survivors during the elitism process. The population for the next generation of the affinity maturation process is the combination of the children and the survivors. Additionally, to prevent overfitting, breeding ceases when the population for a specific class achieves 100% accuracy at detecting that class. In that case, the next generation consists entirely of survivors.

After breeding is completed, mutation is performed on the next generation. A subset of sensors is selected randomly from the population. A random expressed feature in the sensor's chromosome is selected for mutation. If the feature is binary or discrete, a bit is flipped. If the feature is a real value, the value is altered by a random number.

4.5 Sensor Deployment and Refinement

The lifecycle continues by iterating through the negative selection, training and genetic algorithm phases until a maximum number of generations is reached. At this point, the sensors are considered trained (or refined), although they may not have perfect accuracy for their classification. In the pre-deployment phase, the sensors with a secondary fitness greater than 0.5 will become the live sensors. In the sensor refinement phase, those sensors would replace the existing live sensors, as the "next generation" of sensors. The threshold of secondary fitness may be refined to trade off between covering potential attacks and generating too many false positives.

Live deployment of the sensors could not be tested due to bureaucratic issues obtaining the appropriate authorization for live monitoring of the department network. Since live network traffic could contain personally identifying or confidential information, the campus requires assurance that WCIS will protect such information from unauthorized view before granting authorization. As of this time, the authorization is still pending.

Since this bureaucratic restriction prevented the live deployment of sensors to test the concept, the sensors are instead presented with unlabeled data to see how they perform in a real-world scenario. Any sensor with a secondary fitness less than the above threshold is not used for this phase as it has difficulty distinguishing normal requests from attack requests. The sensors determine if each unlabeled request is an attack or a normal request. If the request is labeled an attack by a sensor, the classification of the sensor is recorded. After passing the unlabeled request past all sensors, the classification with the highest "vote" count is chosen as the class label for the request. Those results are then hand-verified to see their accuracy. The results of testing the sensors against unknown data are given in Section 6.

The bureaucratic restriction also made it difficult to fully test the scalability of the pre-deployment, detection and sensor refinement phases. In particular, this made it difficult to fully implement the back-end processing aspects of the sensor refinement phase, as there were no deployment and back-end systems to communicate between. While WCIS contains the algorithmic components of sensor refinement, the practical aspects of deploying sensors, recording statistics, communicating those statistics back to the back-end system, refining sensors on the back-end system and re-deploying the next generation of sensors could not be fully investigated.

The department is currently in the process of building an isolated network. The sensors can be deployed on the isolated network since the data will be simulated, which means campus authorization is not required. This will allow testing of the sensor refinement phase. Scalability testing can also be conducted. Based on the promising results presented in Section 6, it is expected that WCIS will perform well in a simulated live environment. While this is still not an ideal scenario, it will allow continued development and testing of WCIS while the attempts to get campus authorization for live deployment continue.

Table 4: The special characters used in the fingerprinting method.

Character	Description
%	Used by various encoding methods such as hex encoding
'	Used by SQL injection attacks
+	Interpreted by Microsoft IIS as a space
..	Used in directory traversal attacks
\	Used in directory traversal attacks since URIs contain only /
(	Used in cross site scripting attacks
)	Used in cross site scripting attacks
<	Used in cross site scripting attacks
>	Used in cross site scripting attacks
//	Used in proxy attempts or to exploit an old Apache vulnerability

5 Fingerprinting URIs

In order to adapt the AIS method to detect malicious web server requests in WCIS, the web request data must be converted into a pattern consisting of binary, discrete and real value features. The chromosome in each sensors would then seek to match these features. The features from the web request chosen for WCIS are the Uniform Resource Identifier (URI), the HTTP command (GET, POST, HEAD, etc) and the HTTP version. Additional features from the request, such as headers, referrer, and so on, could also be added as features, although they are not supported at this time in WCIS due to the nature of the Apache logs available for data processing. Due to the restrictions imposed by the campus, WCIS has had to run off of Apache logs rather than the live network and the logs are not always configured to log these features. Additionally, WCIS does not look at the IP address of the client or the return code as it is not concerned with detecting the activities of unique clients or whether an attack failed or succeeded. It is concerned with discovering patterns that indicate a zero-day attack has been attempted.

The HTTP command is converted into a discrete bitmap where each set bit refers to a specific command. For example, bit 0 is set for GET, bit 1 is set for POST and so on. The HTTP protocol is likewise converted into

a discrete bitmap, although it could also be modeled as a real value. The length of the URI is converted into a real value feature. Likewise, the number of variables in the URI is also converted into a real value feature. The URI is then parsed to develop a fingerprint of special characters used in the URI. Table 4 summarizes the special characters modeled in the fingerprint. These characters were chosen based on the whitepapers published online at cgisecurity.com [23, 24] and based on the inspection of later web server attacks. Each special character or character sequence listed in Table 4 is modeled as a real value feature.

Real value features are all modeled as a pair of values: [base, offset]. The sensor will match a URI if the URI value is within the range of base to base+offset. When mutating a real value feature, a random value may be added or subtracted from the base, the offset or both. The base can only be altered by a value from -2 to +2. The offset can only be altered by a value of -4 to +4. This prevents mutation from wildly changing the range that a feature detects.

A sensor is considered to match a web request when all of its expressed features matches the features in the web request. For binary features, the feature matches when the corresponding bit to the feature is set in the sensor. For real value features, a feature matches when its value falls within the range of values in the sensor. Note that the web request may contain additional features that the sensor does not check. The matching is driven by the feature set that the sensor expresses, not the feature set in the request.

6 Experimental Results

WCIS was tested using an attack dataset, a normal dataset and an unknown dataset, as described in Section 4. The attack dataset consists of 179 labeled attacks gathered from Bugtraq, live web server logs and tests run on an un-networked machine. The normal dataset consists of 52977 regular requests gathered from the Lincoln Laboratory DARPA dataset [14] and live web server logs.

Obviously, the preferred method of testing WCIS would have been actual live requests to a web server, as this would best approximation of the real-world performance of WCIS. Unfortunately, as described in previous sections, the regulations at this university have made it difficult to do such testing on live web servers due to privacy concerns. Instead, the Apache `access.log` repository for the Computer Science department web server was used for the unknown dataset. 11659 random requests were pulled from the logs and placed into the unknown dataset.

Besides the datasets, WCIS has many parameters that tune its performance. These parameters are:

`pop` The population size for each classification category. A larger population size creates a larger pool of initial random sensors and thus a greater likelihood of randomly creating a "good" sensor.

`gen` The maximum number of generations for the affinity maturation process. The higher this value is, the more likely it is that affinity maturation can derive "good" sensors even if the random initial sensors are only mediocre.

`xover` The percentage of the next generation that comes from breeding. The remaining percentage of the next generation will be survivors.

`mut` The mutation rate for the next generation. A higher value introduces more random change in each generation, which can be beneficial, harmful or benign.

`thresh` The threshold for affinity when doing negative selection. Sensors with affinity above this threshold are destroyed.

`agree` The number of sensors that must agree a request is an attack before it is labeled as an attack. For classification, `2 * agree` must label an unknown data as an attack before it will be classified.

WCIS was tested with population sizes of 25, 50 and 75 for each classification category. Each sensor in a population is analogous to a rule in an IDS in that it looks for a specific pattern in the web request. Note that the actual total number of sensors tested in each tested run of WCIS was `pop*number_of_classifications`. References to "population size" in this section refers to the number of sensors for each classification category (`pop`), not the total number of sensors tested (`pop*number_of_classifications`).

The maximum number of generations tested were 10, 20, 30, 40 and 50. The mutation rates tested were 1%, 2.5%, 5% and 10%. The value for `xover` was 0.6, the value for `threshold` was 0.0002 and the value for `agree` was 3, as prior testing has shown these values yield good results.

6.1 Runtime

One of the first concerns with any method that uses evolutionary computation, such as genetic algorithms, is how long it takes the algorithm to complete. This is one of the motivations behind separating the operation of WCIS into phases: pre-deployment, detection and sensor refinement. Only the pre-deployment and sensor refinement phases will need to run the genetic algorithm.

To test the runtime for the pre-deployment phase, WCIS was tested on a Xeon E5410 2.33GHz machine

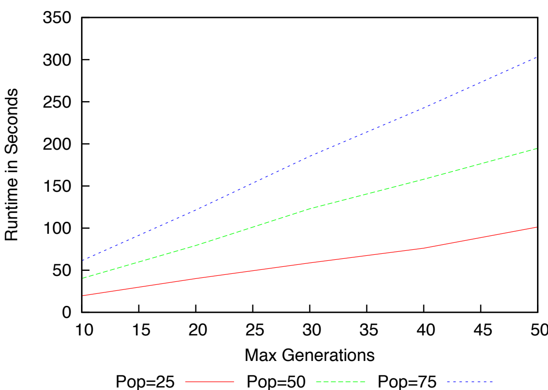


Figure 1: Average runtime for the pre-deployment phase of WCIS for the three tested population sizes for each classification label. Note that the actual total number of sensors is $pop * 6$ since there were 6 classification labels tested. This is purely the pre-deployment phase runtime, not the detection or sensor refinement phase runtime. The detection phase runtime was 0.23 to 0.61 seconds.

with 4GB of RAM. The pre-deployment phase was coded as a single-threaded process. The population for each classification label was processed in a series, using round-robin scheduling (e.g. it processed the first generation of the info class, then the first generation of the traversal class, and so on). With sufficient memory to hold multiple copies of the normal and attack datasets, WCIS could easily be changed to a multi-threaded program with a thread for each population, which would lead to a substantial decrease in the runtime and increase in scalability. The current prototype was also coded in C++, which could be changed to a more efficient programming language in future versions to provide additional scalability.

These changes were not made because the point of this test was to run the genetic algorithm under less than ideal conditions to illuminate choke-points in the underlying algorithms. These choke-points might not be apparent if the code runs too quickly for any differences between input parameters to become significant. This might leave inefficient areas in the underlying algorithms that could affect future scalability. Additionally, if the runtime for WCIS is reasonable under these less than ideal, and easily remedied, coding conditions, then we can be reasonably assured that there are not choke-points in the underlying algorithms.

As shown in Figure 1, even with the largest population sizes and number of generations, WCIS trained the sensors in the pre-deployment phase in under six minutes. This is very reasonable for an evolutionary algorithm, so it is unlikely that there are hidden scalability issues in the underlying algorithms. Converting WCIS to a multi-threaded program in a more efficient programming language should yield even faster results. The sensor refinement phase is expected to have a similar runtime as it needs to run through a similar lifecycle. These results also emphasize why it is important to separate off the evolutionary phases as back-end processes on a separate system from the deployment system. It would be unacceptable to wait 6 minutes for the sensors to refine themselves on a live system, but the separation allows the deployed sensors to continue monitoring live traffic while the back-end system refines the sensors.

While it was not possible at this time to test the detection phase with live data due to the previously described issues, presenting WCIS with the 11659 unknown requests to emulate the detection phase took from 0.23 to 0.61 additional seconds on average, including the extra I/O time to load the unknown dataset from disk, log classifications and log the classification statistics that are presented in the remaining results. There seemed to be little correlation between population size and the additional time required for WCIS to test the unknown requests. For example, the population size of 50 had the lowest average time, while the population size of 25 had the highest average time. This suggests most of variance in the time to test the unknown dataset was due to I/O latency, particularly since the test system had only a consumer-grade SATA drive.

More testing will need to be done to determine the realistic traffic rates that WCIS can handle during the detection phase. These can be conducted once the department's isolated network is completed.

6.2 Accuracy at Classification

Since the primary fitness function was the accuracy at classifying the attack dataset, let us look at the best accuracy for each population in the test runs. Five separate populations for each classification label were tested for each possible combination of variables. The best performing population for each classification and combination was examined.

The best performing populations when the population size was 25 had a maximum number of generations of 40 and a mutation rate of 1%, as shown in Figure 2. The small population size means that WCIS starts with less random diversity. This means the affinity of the initial sensors might be quite low for their desired classification, whereas with a larger population there is a higher chance of randomly generating an antibody with moderate to strong affinity for the class. Because of this low affinity in early generations, the small population size needs more generations for affinity maturation. In par-

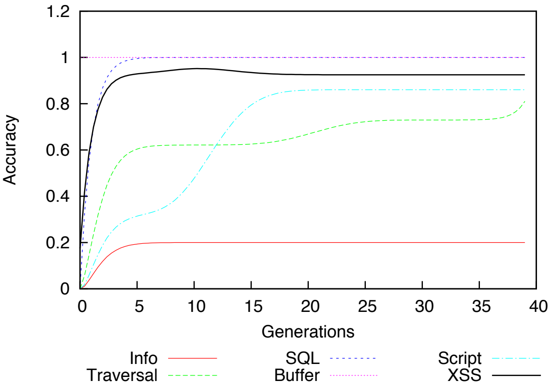


Figure 2: Detection accuracy for each class when the population size for each classification is 25, the maximum generations is 40 and the mutation rate is 1%. This was the best performing population when the population size for each classification was 25.

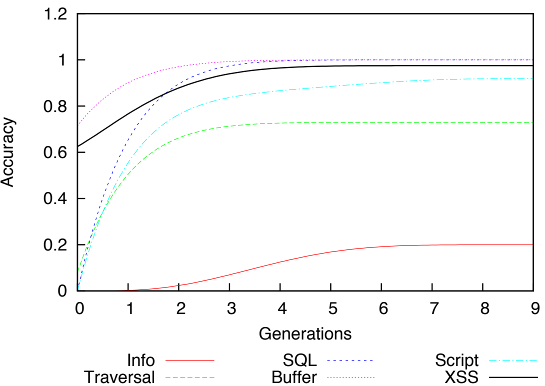


Figure 3: Detection accuracy for each class when the population size for each classification is 50, the maximum generations is 10 and the mutation rate is 2.5%. This was the best performing population when the population size for each classification was 50.

ticular, Figure 2 shows that the "script" and "traversal" classes took the longest number of generations to plateau in accuracy. However, increasing the maximum generation to 50 actually led to overfitting, where the fitness started to decrease in the final generations. This population size also needed the lowest mutation rate of the best tested population sizes. While mutation can help increase the likelihood that the appropriate feature(s) for that classification are affected in a beneficial way, there is also the possibility that mutation might negatively affect the accuracy. A small population is less able to recover from a negative mutation than a large population.

The best performing populations when the population size was 50 had a maximum number of generations of 10 and a mutation rate of 2.5%, as shown in Figure 3. Since WCIS starts off with a larger random population, it is better able to withstand negative mutations and a higher mutation rate can also increase the likelihood of beneficial mutations. This population size also does not need as many generations to achieve good accuracy at classification since it starts with a larger random pool of sensors and there is a greater likelihood of a good sensor being randomly generated in the initial generation. As with a population size of 25, too many generations led to overfitting and a decrease in accuracy, as shown in Figure 4 where the maximum number of generations is 30.

The best performing populations when the population size was 75 had a maximum number of generations of 20 and a mutation rate of 5%, as shown in Figure 5. While most of the classification accuracies plateaued in early generations, the slightly higher rate of mutation allowed for the "info" and "traversal" classifications to randomly find the right combination of features to increase accuracy in later generations. As noted with the other population sizes, a higher number of maximum generations led to overfitting.

Several trends were noticed across all combinations of variables tested. First, regardless of population size, maximum generations and mutation rates, the populations had great difficulty correctly identifying the "info" class of attacks, as shown in Figures 2 through 4. This is not surprising as the "info" class is the hardest to distinguish from normal data. Information gathering attacks are also hard to distinguish from innocent mistakes, such as a typo in the URI.

Second, as noted above, overfitting and loss of accuracy is seen in all tested combinations of variables when the number of generations is high. This is a general problem in single-objective, single-crossover genetic algorithms. This is caused by a loss of diversity within the population. In essence, the sensors become too specialized for specific attack instances and lose the ability to detect more generalized attacks or attacks which lay on the peripheral of the non-self space. It may be the case that another genetic algorithm would be better suited to this problem domain. For example, a multi-objective genetic algorithm, such as NSGA-II [6, 7], is designed to maintain diversity by balancing multiple fitness objectives.

Overall however, the classification scheme employed by WCIS achieves a high rate of accuracy, particularly in the classifications with a large set of attack instances such as "traversal", "script" and "xss". While no population was able to obtain 100% accuracy in those categories, this may be due to the diversity issue. Even so, the accuracy for "traversal" was 81% in many popula-

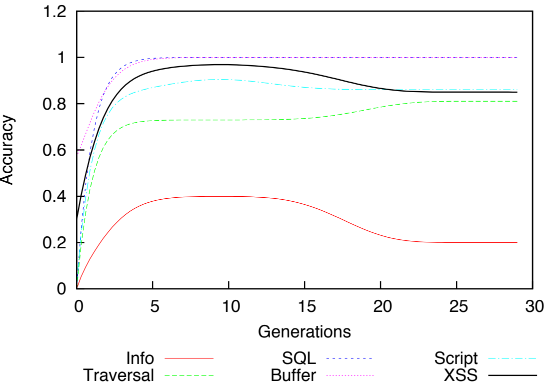


Figure 4: Detection accuracy for each class when the population size for each classification is 50, the maximum generations is 30 and the mutation rate is 2.5%. Note that the extra generations do not yield better results than Figure 3. In fact, overfitting occurs within several classification populations.

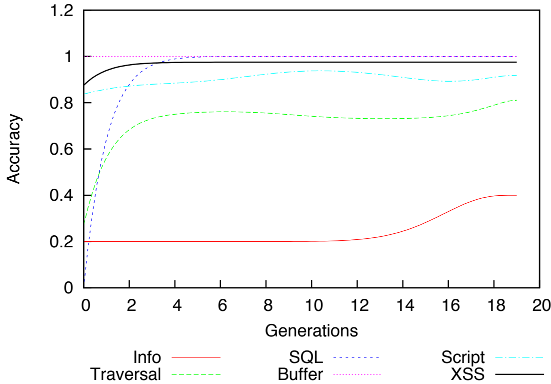


Figure 5: Detection accuracy for each class when the population size for each classification is 75, the maximum generations is 20 and the mutation rate is 5%. This combination of variables had the best classification accuracies of all tested parameters.

tions, the accuracy for "script" was 92% in many populations and the accuracy for "xss" was 92 – 97% in many populations.

6.3 Labeling Unknown Data

After inspecting the accuracy rates, next let us look at how well WCIS could label unknown data gleaned from Apache access logs. The access logs for the Computer Science web server are rotated on a monthly basis, with data going back for several years. Random entries were selected out of two months of access logs. This created an unknown dataset with 11659 entries in it.

After each population finished affinity maturation, it was presented this dataset to label. This emulated a live scan of web traffic. While this test was sufficient to evaluate the effectiveness of WCIS at detecting zero-day attacks, it does not provide metrics for the scalability of WCIS. That would require live testing on networks with various traffic capacities. Unfortunately, due to the previously described challenges with conducting this research in our campus environment, that was not possible at this time. So this test purely focuses on gauging WCIS's ability to detect zero-day attacks and attack variants and its false positive rate when given a large dataset of unlabeled web requests.

It quickly became apparent when looking at the alerts that WCIS raised that someone had tried to attack the web server repeatedly during the time frame covered by the Apache logs. Table 5 shows a subset of the attacks detected by the best population of size 25. Table 6 shows a subset of the attacks detected by the best population of size 50. Finally, Table 7 shows a subset of the attacks detected by the best performing population of size 75.

As seen in the sample of detected attacks, someone attempted to access the /proc/self/environ file, which contains a list of environmental variables, using various directory traversal attempts. This particular attack is actually associated with getting a shell on poorly configured web servers using a combination of directory traversals and shellcode or shell commands injected via the User-Agent field. Even though WCIS does not include the User-Agent field in its feature set, and thus didn't see the actual shellcode that was attempted, it still detected this attack as it appeared in the unknown dataset.

WCIS did have difficulty deciding whether this attack was a "traversal" or a "script" attack since this attack uses directory traversals to access /proc/self/environ to execute code. The attack dataset does classify such attacks as "script" even though they contain features of a "traversal". As pointed out in Table 1, only the attacks which were read-only (such as retrieving the password file) were labeled as "traversal" in the attack dataset. If the directory traversal resulted in an attempt to execute code, it was labeled as a "script" in the attack dataset. Thus, it is not unexpected to see that WCIS has difficulty determining if these attacks were a "script" or a "traversal" since its feature set does not, as of now, include the portion of the attack (the User-Agent field) that would have made it clear it was a "script" attack.

Additionally, looking at the voting data, the attacks labeled as "traversal" also had votes for "script", with many cases having only a difference of one or two votes between the two labels. So the "script" population was detecting these attacks, just not quite as vigorously as the

Table 5: A sample of unknown requests detected as attacks in the unknown dataset for the population in Figure 2.

Class	URL
script	GET /*.php?option=com_dump&controller=..//..//..//..//..//..//..//..///proc/self/environ%0000 HTTP/1.1
traversal	GET /.php?index=../../../../../../../../../../../../../../../proc/self/environ%00 HTTP/1.1
traversal	GET /courses/ls290//index.php?p=../../../../../../../../../../../../../../..//proc/self/environ%0000 HTTP/1.1

Table 6: A sample of unknown requests detected as attacks in the unknown dataset for the population in Figure 3.

Class	URL
script	GET /////?option=com_dump&controller=..//proc/self/environ%0000 HTTP/1.1
script	GET /cs150/index.php?p=../../ HTTP/1.1
traversal	GET /*.php?option=com_dump&controller=..//..//..//..//..//..//..//..///proc/self/environ%0000 HTTP/1.1

"traversal" population.

Being able to detect attacks is desirable, but one also wants an IDS to have a low rate of false alarms. WCIS did not falsely alarm on any of the normal requests in the unknown dataset. This may be due to the fact that some of the requests in the normal dataset were also gleaned from the department Apache access logs. However, this is a good result since it shows that WCIS is easily tuned to the normal traffic for a specific website by using a sampling of that normal traffic to generate the normal dataset.

7 Conclusions

This paper presented a method of detecting zero-day attacks on web servers via malicious requests that is based on artificial immune systems. This prototype system, called Web Classifying Immune System (WCIS), is intended to augment the capabilities of an existing intrusion detection system (IDS) by detecting attacks that are not detectable by the existing IDS. WCIS is a modified artificial immune system (AIS) that adds classification. WCIS also seeks to improve the efficiency of an AIS by separating tasks into the pre-deployment phase, detection phase and sensor refinement phase instead of requiring all these tasks to take place within a single AIS lifecycle. This allows the detection phase to focus on low-resource, speedy sensors while the more costly evolutionary computation associated with the other phases occurs on a separate back-end system.

Notably, WCIS is able to achieve a high rate of accuracy at detecting most classes of attacks in the attack dataset, with the exception of the "info" attacks, which are difficult to distinguish from normal requests. When tested against unlabeled data from Apache access logs, WCIS is able to identify attacks within the requests without falsely alerting on normal traffic. WCIS does have some difficulty choosing between the "traversal" and "script" classifications when the "script" attack uses some elements of directory traversal in its URI. This is likely due to the fact that WCIS only models the HTTP method, URI and HTTP protocol. However, even with this limitation, WCIS is able to detect that an attack containing elements of a directory traversal has occurred.

In summary, WCIS is able to achieve a high rate of accuracy at detecting and classifying attacks against web servers without falsely alarming on normal traffic when properly trained on the normal traffic patterns of the network. WCIS can be easily trained on the normal traffic patterns by giving it a sampling of web server logs, such as Apache logs. The ability to classify the attacks is particularly noteworthy as it allows an administrator to rapidly focus on the initial mitigation and response techniques. It might also lead to integration with an automated response engine, although that has not yet been explored for WCIS.

8 Future Work

The next phase of development for WCIS will focus on creating an appropriate test bed. The department has recently secured a Department of Education grant that is funding the expansion of research laboratory space. A portion of this grant is being used to develop an isolated network. This can be used to test WCIS (and other security tools) without concern about running afoul of the campus privacy regulations. This is not a perfect solution, as it will still be a simulated environment instead of a live environment, but it will permit the full testing of the sensor refinement phase, which has been hampered by the campus regulations. This will also allow scalability testing, although the isolated network funding currently limits the test bed to Gigabit Ethernet instead of 10 Gigabit Ethernet, so there will be limitations to testing the scalability to high capacity networks.

Table 7: A sample of unknown requests detected as attacks in the unknown dataset for the population in Figure 5.

Class	URL
script	GET /.../ports_labeled.jpg HTTP/1.1
traversal	GET /index2.php?option=com_dump&controller=..//..//..//..//..//..//..//..///proc/self/environ%0000 HTTP/1.1
traversal	GET /index.php?t=1&p=technical_info/howto//index.php?filename=../../../../../../../../../../../../../../../proc/self/environ%00 HTTP/1.1
script	GET /faculty/interests/..\\index.html HTTP/1.1

Another area of future development is expanding the feature set used by WCIS sensors. Currently, the feature set of WCIS only models the request line from the request, consisting of the HTTP method, URI and HTTP version. It does not model the general headers, request headers, entity headers or message body specified by Hypertext Transfer Protocol version 1.1 for the HTTP request. This limitation arose because WCIS had to be run on Apache access logs, instead of live data, due to policy restrictions at the institution. The available Apache logs did not consistently record any header fields. However, attackers are using the header fields as a part of their attacks so WCIS should expand its feature set to model these aspects of malicious web server requests. The isolated network test bed should enable the incorporation of these fields into the sensor feature set, since WCIS will no longer be constrained by the formatting of the Apache logs.

Additionally, as noted in Section 6, the genetic algorithm currently being used by WCIS may not be the best algorithm for this problem domain. It suffers from a loss of diversity, which leads to overfitting and a decreased accuracy at detecting and classifying attacks as the generations progress. Another avenue of future research is to explore how other genetic algorithms such as multi-objective genetic algorithms can improve diversity in the sensor population. This diversity will also be useful in detecting novel attacks that do not clearly fall under one of the existing classification categories.

References

[1] BALTHROP, J., ESPONDA, F., FORREST, S., AND GLICKMAN, M. Coverage and generalization in an artificial immune system. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2002)* (July 2002), pp. 3 – 10.

[2] DANFORTH, M. *Models for Threat Assessment in Networks*. PhD thesis, University of Califonia, Davis, Davis, CA, USA, June 2006.

[3] DANFORTH, M. Towards a Classifying Artificial Immune System for Web Server Attacks. In *Proceedings of the International Conference on Machine Learning a nd Applications (ICMLA 2009)* (Miami, FL, USA, December 2009).

[4] DANFORTH, M., AND LEVITT, K. Immune System Model for Detecting Web Server Attacks. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA 2003)* (Los Angeles, CA, USA, June 2003), pp. 161 – 167.

[5] DASGUPTA, D., YU, S., AND MAJUMDAR, N. S. MILA - multilevel immune learning algorithm. In *Proc. Genetic and Evoluntionary Computation Conference (GECCO 2003)* (Chicago, IL, USA, July 2003), pp. 183 – 194.

[6] DEB, K., AGRAWAL, S., PRATAB, A., AND MEYARIVAN, T. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In *Proceedings of the Parallel Problem Solving from Nature VI Conference* (Paris, France, 2000), Springer. Lecture Notes in Computer Science No. 1917, pp. 849–858.

[7] DEB, K., PRATAB, A., AGRAWAL, S., AND MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation 6*, 2 (April 2002), 182 – 197.

[8] FARMER, J. D., PACKARD, N. H., AND PERELSON, A. S. The immune system, adaptation, and machine learning. In *Physica D* (1986), pp. 22:187 – 204.

[9] FORREST, S., PERELSON, A. S., ALLEN, L., AND CHERUKURI, R. Self-nonself discrimination in a computer. In *Proc. 1994 IEEE Symposium on Research in Security and Privacy* (Los Alamitos, CA, USA, 1994), pp. 202 – 214.

[10] GONZALES, F., DASGUPTA, D., AND GOMEZ, J. The effect of binary matching rules in negative selection. In *Proc. Genetic and Evoluntionary Computation Conference (GECCO 2003)* (Chicago, IL, USA, July 2003), pp. 195 – 206.

[11] GONZALEZ, F., DASGUPTA, D., AND KOZMA, R. Combining negative selection and classification techniques for anomaly detection. In *Proc. 2002 Congress on Evoluntionary Computation (CEC 2002)* (Honolulu, HI, USA, May 2002), pp. 705 – 710.

[12] HOFMEYER, S., AND FORREST, S. Architecture for an artificial immune system. *Evolutionary Computation Journal 8*, 4 (2000), 433 – 473.

[13] HOFMEYER, S. A., AND FORREST, S. Immunity by design: An artificial immune system. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)* (San Francisco, CA, USA, 1999), pp. 1289 – 1296.

[14] MIT LINCOLN LABORATORY. Darpa intrusion detection evaluation, 1999. http://www.ll.mit.edu/IST/ideval/.

[15] SECURITYFOCUS. Bugtraq mailing list, 2002 – present. http://www.securityfocus.com/archive/1.

[16] SOMAYAJI, A., HOFMEYER, S., AND FORREST, S. Principles of a computer immune system. In *Proc. New Security Paradigms Workshop (NSPW-97)* (Langdale, UK, 1997), pp. 75 – 82.

[17] WATKINS, A. *A resource limited artificial immune classifier*. PhD thesis, Mississippi State University, 2001.

[18] WATKINS, A., AND BOGGESS, L. A new classifier based on resource limited artificial immune systems. In *IEEE Congress on Evolutionary Computation* (Honolulu, HI, USA, May 2002), pp. 1546 – 1551.

[19] WATKINS, A., AND BOGGESS, L. A resource limited artificial immune classifier. In *IEEE Congress on Evolutionary Computation* (Honolulu, HI, USA, May 2002), pp. 926 – 931.

[20] WATKINS, A., AND TIMMIS, J. Artificial Immune Recognition System (AIRS): Revisions and refinements. In *Proceedings of the 1st International Conference on Artificial Immune Systems (ICARIS)* (University of Kent at Canterbury, UK, 2002), pp. 173 – 181.

[21] WATKINS, A., TIMMIS, J., AND BOGGESS, L. Artificial Immune Recognition System (AIRS): An Immune-Inspired Supervised Learning Algorithm. *Genetic Programming and Evolvable Machines 5*, 3 (2004), 291 – 317.

[22] WILLIAMS, P. D., ANCHOR, K. P., BEBO, J. L., GUNSCH, G. H., AND LAMONT, G. D. CDIS: Towards a computer immune system for detecting network intrusions. In *Proc. Recent Advances in Intrusion Detection (RAID 2001)* (Davis, CA, USA, October 2001), pp. 117 – 133.

[23] ZENOMORPH (ADMIN@CGISECURITY.COM). Fingerprinting port 80 attacks: A look into web server, and web application attack signatures. Whitepaper, November 2001. http://www.cgisecurity.com/papers/fingerprint-port80.txt.

[24] ZENOMORPH (ADMIN@CGISECURITY.COM). Fingerprinting port 80 attacks: A look into web server, and web application attack signatures: Part 2. Whitepaper, March 2002. http://www.cgisecurity.com/papers/fingerprinting-2.txt.

Automating Network and Service Configuration Using NETCONF and YANG

Stefan Wallin
Luleå University of Technology
stefan.wallin@ltu.se

Claes Wikström
Tail-f Systems AB
klacke@tail-f.com

Abstract

Network providers are challenged by new requirements for fast and error-free service turn-up. Existing approaches to configuration management such as CLI scripting, device-specific adapters, and entrenched commercial tools are an impediment to meeting these new requirements. Up until recently, there has been no standard way of configuring network devices other then SNMP and SNMP is not optimal for configuration management. The IETF has released NETCONF and YANG which are standards focusing on Configuration management. We have validated that NETCONF and YANG greatly simplify the configuration management of devices and services and still provide good performance. Our performance tests are run in a cloud managing 2000 devices.

Our work can help existing vendors and service providers to validate a standardized way to build configuration management solutions.

1 Introduction

The industry is rapidly moving towards a service-oriented approach to network management where complex services are supported by many different systems. Service operators are starting a transition from managing pieces of equipment towards a situation where an operator is actively managing the various aspects of services. Configuration of the services and the affected equipment is among the largest cost-drivers in provider networks [9]. Delivering valued-added services, like MPLS VPNS, Metro Ethernet, and IP TV is critical to the profitability and growth of service providers. Time-to-market requirements are critical for new services; any delay in configuring the corresponding tools directly affects deployment and can have a big impact on revenue. In recent years, there has been an increasing interest in finding tools that address the complex problem of deploying service configurations. These tools need to replace the current configuration management practices that are dependent on pervasive manual work or ad hoc scripting. Why do we still apply these sorts of blocking techniques to the configuration management problem? As Enck [9] points out, two of the primary reasons are the variations of services and the constant change of devices. These underlying characteristics block the introduction of automated solutions, since it will take too much time to update the solution to cope with daily changes. We will illustrate that a NETCONF [10] and YANG [4] based solution can overcome these underlying challenges.

Service providers need to be able to dynamically adopt the service configuration solutions according to changes in their service portfolio without defining low level device configuration commands. At the same time, we need to find a way to remove the time and cost involved in the plumbing of device interfaces and data models by automating device integration. We have built and evaluated a management solution based on the IETF NETCONF and YANG standards to address these configuration management challenges. NETCONF is a configuration management protocol with support for transactions and dedicated configuration management operations. YANG is a data modeling language used to model configuration and state data manipulated by NETCONF. NETCONF was pioneered by Juniper which has a good implementation in their devices. See the work by Tran [23] et. al for interoperability tests of NETCONF.

Our solution is characterized by the following key characteristics:

1. *Unified YANG modeling* for both services and devices.
2. One database that *combines device configuration and service configuration*.
3. *Rendering* of northbound and southbound interfaces and database schemas from the service and device model. Northbound are the APIs published to users

of NCS, be it human or programmatic interfaces. Southbound is the integration point of managed devices, for example NETCONF.

4. A *transaction engine* that handles transactions from the service order to the actual device configuration deployment.

5. An *in-memory high-performance database*.

To keep the service and device model synchronized, (item 1 and 2 above), it is crucial to understand how a specific service instance is actually configured on each network device. A common problem is that when you tear down a service you do not know how to clean up the configuration data on a device. It is also a well-known problem that whenever you introduce a new feature or a new network device, a large amount of glue code is needed. We have addressed this again with annotated YANG models rather then adaptor development. So for example, the YANG service model renders a northbound CLI to create services. From a device model in YANG we are actually able to render the required Cisco CLI commands and interpret the response without the need for the traditional Perl and Expect scripting. Currently our solution can integrate without any plumbing.

It is important to address the configuration management problem using a transactional approach. The transaction should cover the whole chain including the individual devices. Finally, in order to manipulate configuration data for a large network and many service instances we need fast response to read and write operations. Traditional SQL and file-based database technologies fall short in this category. We have used an in-memory database journaled to disk in order to address performance and persistence at the same time.

The objectives of this research are to determine whether these new standards can help to eliminate the device integration problem and provide a service configuration solution utilizing automatically integrated devices. We have studied challenges around data-model discovery, interface versioning, synchronization of configuration data, multi-node configuration deployment, transactional models, and service modeling issues. In order to validate the approach we have used simulated scenarios for configuring load balancers, web servers, and web sites services. Throughout the use-cases we also illustrate the possibilities for automated rendering of Command Line interfaces as well as User Interfaces from YANG models.

Our studies show that a NETCONF/YANG based configuration management approach removes unnecessary manual device integration steps and provides a platform for multi-device service configurations. We see that problems around finding correct modules, loading them and creating a management solution can largely be automated. In addition to this, the transaction engine in our solution combined with inherent NETCONF transaction capabilities resolves problems around multi-device configuration deployment.

We have run performance tests with 2000 devices in an Amazon cloud to validate the performance of NETCONF and our solution. Based on these tests we see that the solution scales and NETCONF provides a configuration management protocol with good performance.

2 Introduction to NETCONF and YANG

The work with NETCONF and YANG started as a result of an IAB workshop held in 2002. This is documented in RFC 3535 [18].

> "*The goal of the workshop was to continue the important dialog started between network operators and protocol developers, and to guide the IETFs focus on future work regarding network management*."

The workshop concluded that SNMP is not being used for configuration management. Operators put forth a number of requirements that are important for a standards-based configuration management solution. Some of the requirements were:

1. Distinction between configuration data and data that describes operational state and statistics.

2. The capability for operators to configure the network as a whole rather than individual devices.

3. It must be easy to do consistency checks of configurations.

4. The availability of text processing tools such as diff, and version management tools such as RCS or CVS.

5. The ability to distinguish between the distribution of configurations and the activation of a certain configuration.

NETCONF addresses the requirements above. The design of NETCONF has been influenced by proprietary protocols such as Juniper Networks JUNOScript application programming interface [14].

For a more complete introduction see the Communications Magazine article [19] written by Schönwälder et al.

2.1 NETCONF

The Network Configuration Protocol, NETCONF, is an IETF network management protocol and is published in RFC 4741. NETCONF is being adopted by major network equipment providers and has gained strong industry support. Equipment vendors are starting to support NETCONF on their devices, see the NETCONF presentation by Moberg [16] for a list of public known implementations.

NETCONF provides mechanisms to install, manipulate, and delete the configuration of network devices. Its operations are realized on top of a simple Remote Procedure Call (RPC) layer. The NETCONF protocol uses XML based data encoding for the configuration data as well as the protocol messages. NETCONF is designed to be a replacement for CLI-based programmatic interfaces, such as Perl + Expect over Secure Shell (SSH). NETCONF is usually transported over the SSH protocol, using the "NETCONF" sub-system and in many ways it mimics the native proprietary CLI over SSH interface available in the device. However, it uses structured schema-driven data and provides detailed structured error return information, which the CLI cannot provide.

NETCONF has the concept of logical data-stores such as "writable-running" or "candidate" (Figure 1). Operators need a way to distribute changes to the devices and validate them locally before activating them. This is indicated by the two bottom options in Figure 1 where configuration data can be sent to candidate databases in the devices before they are committed to running in production applications.

All NETCONF devices must allow the configuration data to be locked, edited, saved, and unlocked. In addition, all modifications to the configuration data must be saved in non-volatile storage. An example from RFC 4741 that adds an interface named "Ethernet0/0" to the running configuration, replacing any previous interface with that name is shown in Figure 2.

2.2 YANG

YANG is a data modeling language used to model configuration and state data. The YANG modeling language is a standard defined by the IETF in the NETMOD working group. YANG can be said to be tree-structured rather than object-oriented. Configuration data is structured into a tree and the data can be of complex types such as lists and unions. The definitions are contained in modules and one module can augment the tree in another module. Strong revision rules are defined for modules. Figure 3 shows a simple YANG example. YANG is mapped to a NETCONF XML representation on the wire.

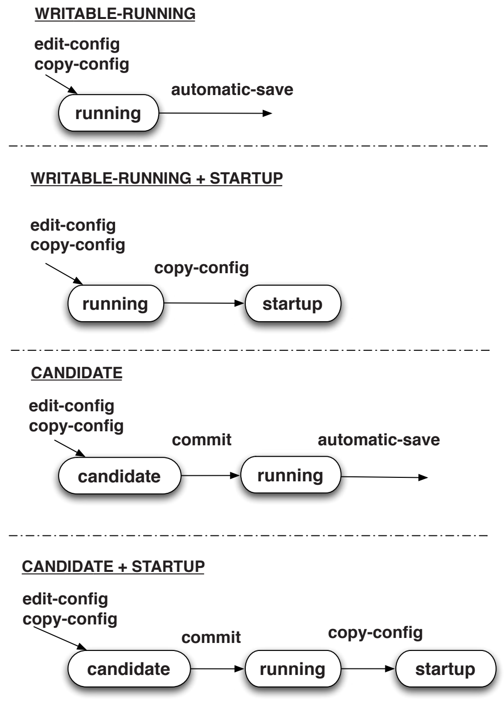


Figure 1: NETCONF Datastores

```
<rpc message-id="101"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <edit-config>
   <target>
      <running/>
   </target>
    <config
    xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
       <top xmlns="http://example.com/schema/1.2/config">
          <interface xc:operation="replace">
            <name>Ethernet0/0</name>
               <mtu>1500</mtu>
               <address>
                 <name>192.0.2.4</name>
                 <prefix-length>24</prefix-length>
              </address>
          </interface>
       </top>
    </config>
 </edit-config>
</rpc>

<rpc-reply message-id="101"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

Figure 2: NETCONF edit-config Operation

```
module acme-system {
  namespace
  "http://acme.example.com/system";
  prefix "acme";
  organization "ACME Inc.";
  contact "joe@acme.example.com";
  description
    "The ACME system.";
  revision 2007-11-05 {
      description "Initial revision.";
  }
  container system {
    leaf host-name {
        type string;
    }
    leaf-list domain-search {
      type string;
    }
    list interface {
      key "name";
      leaf name {
          type string;
      }
      leaf type {
          type enumeration {
              enum ethernet;
              enum atm;
            }
        }
      leaf mtu {
          type int32;
      }
      must ‘‘ifType != ’ethernet’ or ‘‘+
      ‘‘(ifType = ’ethernet’ and ‘‘ +
      ‘‘mtu = 1500)‘‘ {
      }
...
```

Figure 3: YANG Sample

YANG also differs from previous network management data model languages through its strong support of constraints and data validation rules. The suitability of YANG for data models can be further studied in the work by Xu et. al [24].

3 Our Config Manager Solution - NCS

3.1 Overview

We have built a layered configuration solution, NCS, Network Configuration Server. See Figure 4. The *Device Manager* manages the NETCONF devices in the network and heavily leverage the features of NETCONF and YANG to render a Configuration Manager from the YANG models. At this layer, the YANG models represent the capabilities of the devices and NCS provides the device configuration management capabilities.

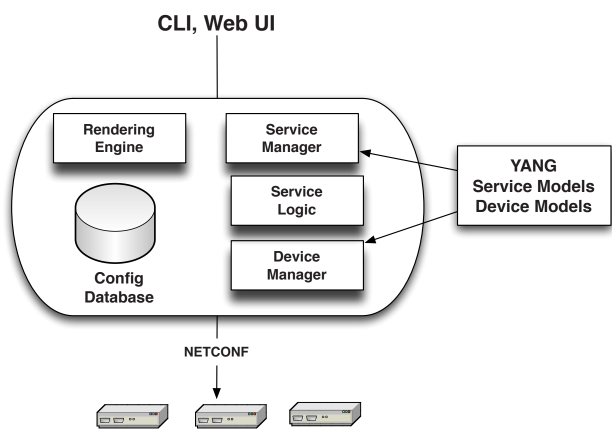


Figure 4: NCS - The Configuration Manager

The *Service Manager* in turn lets developers add YANG service models. For example, it is easy to represent end-to-end connections over L2/L3 devices or web sites utilizing load balancers and web servers. The most important feature of the Service Manager is to transform a service creation request into the corresponding device configurations. This mapping is expressed by defining service logic in Java which basically does a model transformation from the service model to the device models.

The *Configuration Database, (CDB)*, is an in-memory database journaled to disk. CDB is a special-purpose database that targets network management and the in-memory capability enables fast configuration validation and performs diffs between running and candidate databases. Furthermore the database schema is directly rendered from the YANG models which removes the need for mapping between the models and for example a SQL database. A fundamental problem in network management is dealing with different versions of device interfaces. NCS is able to detect the device interfaces through its NETCONF capabilities and this information is used by CDB to tag the database with revision information. Whenever a new model revision is detected, NCS can perform a schema upgrade operation. CDB stores the configuration of services and devices and the relationships between them. NETCONF defines dedicated operations to read the configuration from devices and this drastically reduces the synchronization and reconciliation problem.

Tightly connected to CDB is the *transaction manager*

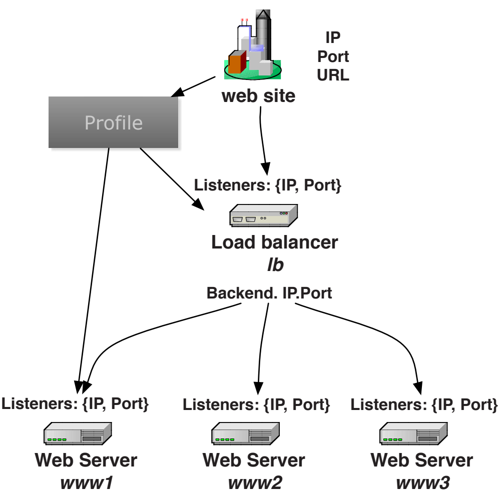


Figure 5: The Example

which manages every configuration change as a transaction. Transactions include all aspects from the service model to all related device changes.

At this point it is important to understand that the NETCONF and NCS approach to configuration management does not use a push and pull approach to versioned configuration files. Rather, it is a fine-grained transactional view based on data models.

The *Rendering Engine* renders the database schemas, a CLI, and a Web UI from the YANG models. In this way the Device Manager features will be available without any coding.

3.2 The Example

Throughout the rest of this paper we will use an example that targets configuration of web-sites across a load balancer and web servers. See Figure 5.

The *service model* covers the aspects of a *web site*; IP Address, Port, and URL. Whenever you provision a web site you refer to a *profile* which controls the selection of load balancers and web servers. A web site allocates a listener on the load balancer which in turn creates backends that refer to physical web servers. So when provisioning a new web site you do not have to deal with the actual load balancer and web server configuration. You just refer to the profile and the service logic will configure the devices. The involved YANG models are :

- `website.yang` : the service model for a web site, it defines web site attributes like url, IP Address, port, and pointer to profile.

- `lb.yang` : the device model for load balancers, it defines listeners and backends where the listeners refers to the web site and backends to the corresponding web servers.

- `webserver.yang` : the device model for a physical web server, it defines listeners, document roots etc.

  The devices in our example are:

  - Load Balancer : `lb`
  - Web Servers : `www1, www2, www3`

3.3 The Device Manager

The Device Manager layer is responsible for configuring devices using their specific data-models and interfaces. The NETCONF standard defines a capability exchange mechanism. This implies that a device reports its supported data-models and their revisions when a connection is established. The capability exchange mechanism also reports if the device supports a `<writable-running>` or `<candidate>` database.

After connection the Device Manager can then use the `get-schema` RPC, as defined in the netconf-monitoring RFC [20] to get the actual YANG models from all the devices. NCS now renders northbound interfaces such as a common CLI and Web UI from the models. The NCS database schema is also rendered from the data-models.

The NCS CLI in Figure 6 shows the discovered capabilities for device “www1”. We see that www1 supports 6 YANG data-models, `interfaces`, `webserver`, `notif`, and 3 standard IETF modules. Furthermore the web-server supports NETCONF features like `confirmed-commit, rollback-on-error` and `validation` of configuration data.

In Figure 7 we show a sequence of NCS CLI commands that first uploads the configuration from all devices and then displays the configuration from the NCS configuration database. So with this scenario we show that we could render the database schema from the YANG models and persist the configuration in the configuration manager.

Now, let’s do some transaction-based configuration changes. The CLI sequence in Figure 8 starts a transaction that will update the ntp server on www1 and the load-balancer. Note that NCS has the concept of a *candidate* database and a *running*. The first represents the desired configuration change and the running database represents the actual configuration of the devices. At the end of the sequence in Figure 8 we use the CLI command `‘‘compare running brief’’` to show the difference between the running and the candidate database. This is what will be committed to the devices. Note that we do a diff and only send the diff. Our in-memory database enables good performance even for large configurations and large networks.

```
ncs> show ncs managed-device www1 capability <RET>
URI                                                    REVISION    MODULE
-----------------------------------------------------------------------------------------------------
candidate:1.0                                          -           -
confirmed-commit:1.0                                   -           -
confirmed-commit:1.1                                   -           -
http://acme.com/if                                     2009-12-06  interfaces
http://acme.com/ws                                     2009-12-06  webserver
http://router.com/notif                                -           notif
rollback-on-error:1.0                                  -           -
urn:ietf:params:netconf:capability:notification:1.0    -           -
urn:ietf:params:xml:ns:yang:ietf-inet-types            2010-09-24  ietf-inet-types
urn:ietf:params:xml:ns:yang:ietf-yang-types            2010-09-24  ietf-yang-types
urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring    2010-06-22  ietf-netconf-monitoring
validate:1.0                                           -           -
validate:1.1                                           -           -
writable-running:1.0                                   -           -
xpath:1.0                                              -           -
```

Figure 6: NETCONF Capability Discovery

```
ncs> request ncs sync direction from-device <RET>
...

ncs> show configuration ncs \
managed-device www1 config <RET>

host-settings {
    syslog {
        server 18.4.5.6 {
            enabled;
            selector 1;
        }
...
}
ncs> show configuration ncs \
managed-device lb config  <RET>

lbConfig {
    system {
        ntp-server 18.4.5.6;
        resolver {
            search     acme.com;
            nameserver 18.4.5.6;
        }
    }
}
```

Figure 7: Synchronize Configuration Data from Devices

In the configuration scenarios shown in Figure 8 we used the auto-rendered CLI based on the native YANG modules that we discovered from the devices. So it gives the administrator one CLI with transactions across the devices, but still with different commands for different

```
ncs% set ncs managed-device \
www1 config host-settings ntp server 18.4.5.7 <RET>

ncs% set ncs managed-device \
lb config lbConfig system ntp-server 18.4.5.7 <RET>

ncs% compare running brief <RET>

 ncs {
     managed-device lb {
         config {
             lbConfig {
                 system {
-                    ntp-server 18.4.5.6;
+                    ntp-server 18.4.5.7;
                 }
...

ncs% commit
```

Figure 8: Configuring two Devices in one Transaction

vendors in case of non-standard modules. NCS allows for device abstractions, where you can provide a generic YANG module across vendor-specific ones.

Every commit in the scenarios described above resulted in a transaction across the involved devices. In this case the devices support the confirmed-commit capability. This means that the manager performs a commit to the device with a time-out. If the device does not get the confirming commit within the time-out period it reverts to the previous configuration. This is also true for restarts or if the SSH connection closes.

3.4 The Service Manager

In our example we have defined a service model corresponding to web-sites and the corresponding service logic that maps the service model to load balancers and web servers. The auto-rendered Web UI let operators create a web site like the one illustrated in Figure 9.

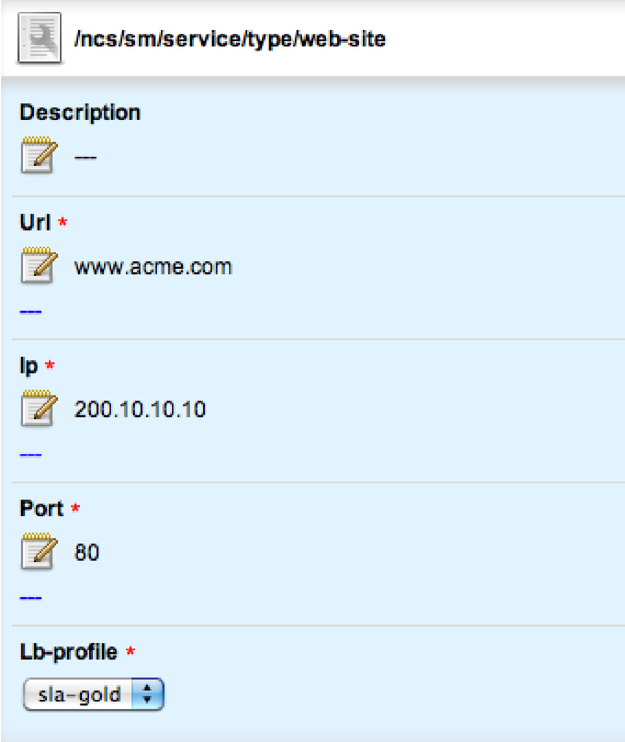


Figure 9: Instantiating a Web-site Service

A fundamental part of the Service Manager is that we use YANG to model services as well as devices. In this way we can ensure that the service model is consistent with the device model. We do this at compile time by checking the YANG service model references to the device model elements. At run-time, the service model constraints can validate elements in the device-model including referential integrity of any references. Let's illustrate this with a simple example. Figure 10 shows a type-safe reference from the web-site service model to the devices. The YANG `leafref` construct refers to a path in the model. The path is verified to be correct according to the model at compile time. At run-time, if someone tries to delete a managed device that is referred to by a service this would violate referential integrity and NCS would reject the operation.

This service provisioning request initiates a hierarchical transaction where the service instance is a parent transaction which fires off child transactions for every

```
leaf lb {
  description "The load balancer to use.";
  mandatory true;
  type leafref {
     path "/ncs:ncs/ncs:managed-device/ncs:name";
   }
}
```

Figure 10: Service-Model Reference to Device-Model

device. In this specific case the selected profile uses all web servers at the device layer. Either the complete transaction succeeds or nothing will happen. As a result the transaction manager stores the resulting device configurations in CDB as shown in Figure 11.

```
Device-modifications

 ncs {
     managed-device lb {
         config {
             lbConfig {
-                listener 200.10.10.10 80 {
-                    service acme-inc {
-                        number 1;
-                        URL-pattern www.acme.com;
-                        backend 192.168.0.9 8008 {
-                        }
-                        backend 192.168.0.10 8008 {
-                        }
-                        backend 192.168.0.11 8008 {
-                        }
-                        session {
-                            type IP;
-                        }
-                    }
-                }
             }
         }
     }
     managed-device www1 {
         config {
             interface eth0 {
-                alias 0 {
-                    ipv4-address 192.168.0.9;
-                }
             }
             wsConfig {
-                listener 192.168.0.9 8008 {
-                }
             }
```

Figure 11: The relationship from a Service to the Actual Device Configurations.

You see that the web-site for `acme` created a listener on the load balancer with backends that maps to the actual web servers. The service also created a listener on the web server. You might wonder why there is a minus-sign for the diff. The reason is that we are actually storing how to delete the service. This means that there will never be any stale configurations in the network. As soon as you delete a service, NCS will automatically clean up.

4 Evaluation

4.1 Performance Evaluation

We have evaluated the performance of the solution using 2000 devices in an Amazon Cloud. The Server is a 4 Core CPU, 4 GB RAM, 1.0 GHz, Ubuntu 10.10 Machine. Here we illustrate 4 test-cases. All test-cases are performed as one single transaction:

1. Start the system with an empty database and upload the configuration over NETCONF from all devices (Figure 12 A).
2. Check if the configuration database is in sync with all the devices (Figure 12 B).
3. Perform a configuration change on all devices (Figure 13 A).
4. Create 500 services instances that touch 2 devices each (Figure 13 B).
5. In Figure 14 we show the memory and database journal disc space for configuring 500 service instances.

All of the test-cases involve the complete transaction including the NETCONF round-trip to the actual devices in the cloud. So, cold-starting NCS and uploading the configuration from 500 devices takes about 8 minutes (Figure 12) and 2000 devices takes about 25 minutes. The configuration synchronization check utilizes a transaction ID to compare the last performed change from NCS to any local changes made to the device. This test assumes that there is some way to get a transaction ID or checksum from the device that corresponds to the last change irrespective of which interface is used. If that is not available and you had to get the complete configuration, then the numbers would be higher.

Updating the config on 500 devices takes roughly one minute, (Figure 8). As seen by Figure 14 the in-memory database has a small footprint even for large networks. In this scenario it is important to note that we always diff the configuration change within NCS before sending it to the device. This means that we only send the actual changes that are needed and this database comparison is included in the numbers. This is an area where we have seen performance bottlenecks in previous solutions when traditional database technologies are used.

These performance tests cover two aspects: performance of NETCONF, and our actual implementation.

NETCONF as a protocol ensures that we achieve at least equal performance to CLI screen scraped solutions and superior performance to SNMP based configuration solutions. XML processing is considerably less CPU intensive than SSH processing.

When running a transaction that touches many managed devices, we use two tricks that affect performance. We pipeline NETCONF RPCs, sending several RPCs in a row, and collecting all the replies in a row. We can also (in parallel) send the requests to all participating managed devices, and then (in parallel) harvest the pipelined replies.

NCS is implemented in Erlang [3, 11] and OTP (Open Telecom Platform) [22] which have excellent support for concurrency and multi-core processors. A lot of effort has gone into parallelizing the southbound requests. For example initial NETCONF SSH connection establishment is done in parallel, greatly enhancing performance.

The network configuration data is kept in a RAM database together with a disk journaling component. If the network is huge, the amount of RAM required can be substantial. When the YANG files are compiled we hash all the symbols in the data models, thus the database is actually a large tree of integers. This increases processing speed and decreases memory footprint of the configuration daemon. The RAM database itself is implemented as an Erlang driver that uses skip lists [17].

Our measurements show that we can handle thousands of devices and hundred thousands of services on off-the-shelf hardware, (4 Core CPU, 4 GB RAM, 1.0 GHz).

We have also made some measurements comparing SNMP and NETCONF performance. We read the interface table using SNMP get-bulk and NETCONF get. In general NETCONF performed 3 times quicker than SNMP. The same kind of performance improvements using NETCONF rather than SNMP can be found in the work by Yu and Ajarmeh [25].

4.2 NETCONF/YANG Evaluation

Let's look at the requirements set forth by RFC 3535 and validate these based on our implementation.

4.2.1 Distinction between configuration data, and data that describes operational state and statistics

This requirement is fulfilled by YANG and NETCONF in that you can explicitly request to get only the configuration data from the device, and elements in YANG are annotated if they are configuration data or not. This greatly simplifies the procedure to read and synchronize configuration data from the devices to a network management system. In our case, NCS can easily synchronize its configuration database with the actual devices.

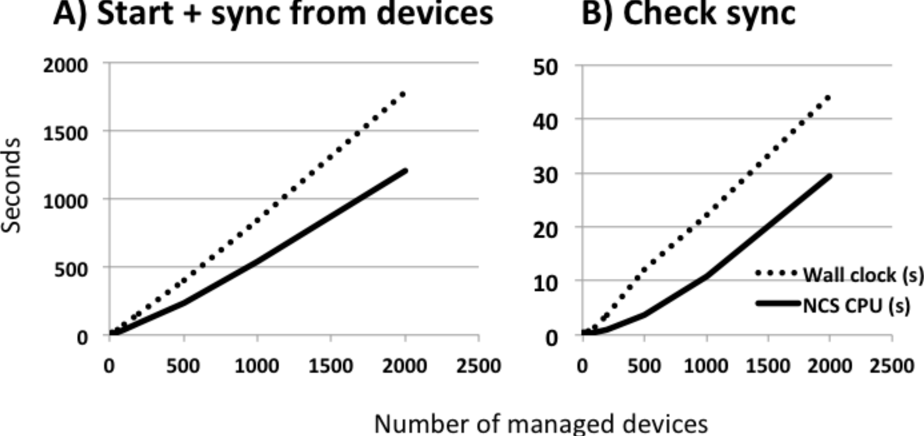


Figure 12: Starting NCS and Reading the Configuration from all Devices
(Dotted line represents Wall Clock Time, Solid Line CPU Time).

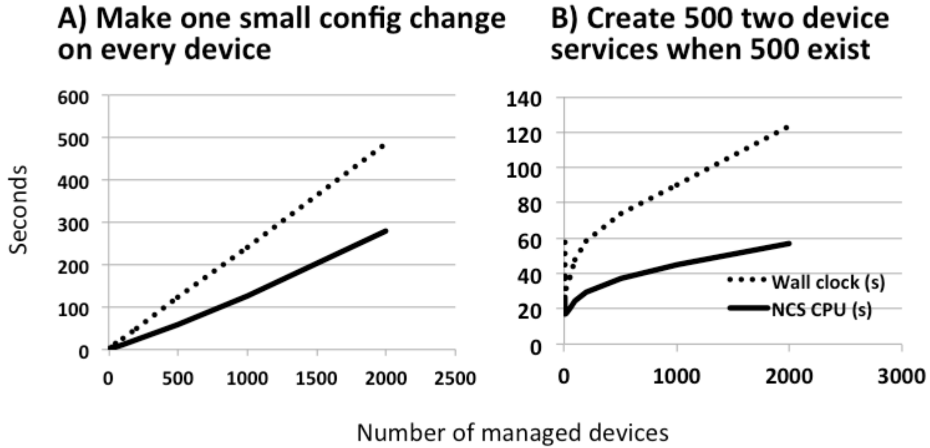


Figure 13: Making Device Configurations and Service Configurations

4.2.2 It is necessary to enable operators to concentrate on the configuration of the network as a whole rather than individual devices

We have validated this from two perspectives

1. Configuring a set of devices as one transaction.

2. Transforming a service configuration to the corresponding device configurations.

Using NCS, we can apply configurations to a group of devices and the transactional capabilities of NETCONF will make sure that the whole transaction is applied or no changes are made at all. The NETCONF `confirmed-commit` operation has proven to be especially useful in order to resolve failure scenarios. A problem scenario in network configuration is that devices may become unreachable after a reconfiguration. The `confirmed-commit` operation requests the device to take the new configuration live but if an acknowledge-

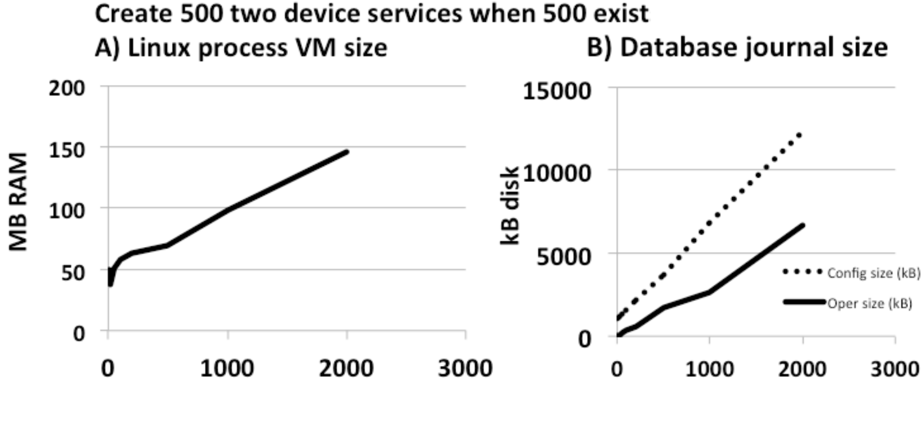


Figure 14: Memory and Journaling Disc Space

ment is not received within a time-out the device automatically rolls-back. This is the way NCS manages to roll-back configurations if one or several of the devices in a transaction does not accept the configuration. It is notable to see the lack of complex state-machines in NCS to do roll-backs and avoid multiple failure scenarios.

In some cases, you would like to apply a global configuration change to all your devices in the network. In the general case the transaction would fail if one of the devices was not reachable. There is an option in NCS to backlog unresponsive devices. In this case NCS will make the transaction succeed and store outstanding requests for later execution.

4.2.3 It must be easy to do consistency checks of configurations.

Models in YANG contain ``must'' expressions that put constraints on the configuration data. See for example Figure 3 where the must expression makes sure that the MTU is set to correct size. So for example, a NETCONF manager can edit the candidate configuration in a device and ask the device to validate it. In NCS we also use YANG to specify service models. In this way we can use must expressions to make sure that a service configuration is consistent including the participating devices. Figure 15 shows a service configuration expression that verifies that the subnet only exists once in the VPN.

4.2.4 It is highly desirable that text processing tools [...] can be used to process configurations.

Since NETCONF operations use well-defined XML payloads, it is easy to process configurations. For example doing a diff between the configuration in the device versus the desired configuration in the management system. The CLI output in Figure 16 shows a diff between a device configuration and the NCS Configuration Database. In this case a system administrator has used local tools on web server 1 and changed the document root, and removed the listener.

```
must  "count(
    ../../mv:access-link[subnet =
    current()/../subnet]) = 1" {
      error-message "Subnet must be unique
      within the VPN";
    }
```

Figure 15: Service Configuration Consistency

4.2.5 It is important to distinguish between the distribution of configurations and the activation of a particular configuration.

The concept of multiple data-stores in NETCONF lets managers push the configuration to a candidate database, validate it, and then activate the configuration by committing it to the running datastore. Figure 17 shows an extract from the NCS trace when activating a new configuration in web server 2.

```
ncs> request ncs managed-device \
www1 compare-config outformat cli <RET>

diff
 ncs {
     managed-device www1 {
         config {
             wsConfig {
                 global {
-                    ServerRoot /etc/doc;
+                    ServerRoot /etc/docroot;
                 }
-                listener 192.168.0.9 8008 {
-                }
             }
         }
     }
 }
```

Figure 16: Comparing Configurations

```
ncs% set ncs managed-device \
www2 config wsConfig global ServerRoot /etc/doc <RET>

ncs% commit | details <RET>
  ncs: SSH Connecting to admin@www2
  ncs: Device: www2 Sending edit-config
  ncs: Device: www2 Send commit
Commit complete.
```

Figure 17: Separation of Distribution of Configurations and Activation

5 Related Work

5.1 Mapping to Taxonomy of Configuration Management Tools

We can map our solution to other Configuration Management solutions based on the taxonomy defined by Delaet and Joosen [7]. They define a taxonomy based on 4 criteria: abstraction level, specification language, consistency, and distributed management.

The abstraction level ranges from high-level end-to-end requirements to low-level bit-requirements. As shown in Figure 18 and described below, in our solution we work with level 1-5 of the 6 mentioned abstraction levels.

1. *End-to-end Requirements* - The service models in the Service Manager expresses end-to-end requirements including constraints expressed as XPATH must expressions. In the case of our web site provisioning example this corresponds to the model for a web site - `website.yang`.

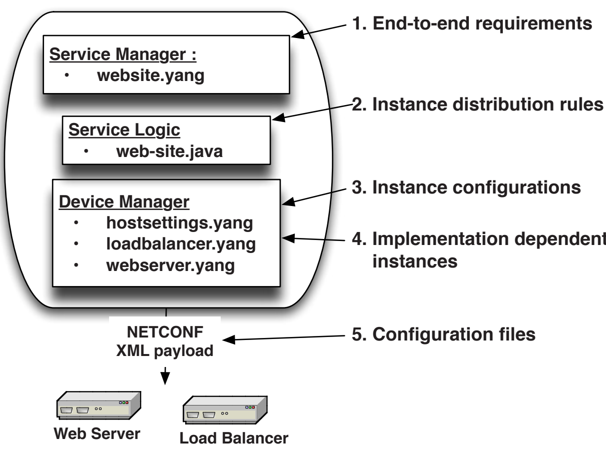


Figure 18: NCS in the Configuration Taxonomy Defined by Delaet and Joosen

2. *Instance Distribution Rules* - How an end-to-end service is allocated to resources is expressed in the Java Service Logic Layer. In this layer we map the provisioning of a web site to the corresponding load balancer and web-server models.

3. Instance Configurations - The changed configuration of devices in the Device Manager. The result of the previous point is a diff, configuration change, sent to NCS Device Manager. The Device Manager has two layers. The device independent layer that can abstract different data-models for the same feature and the concrete device model layer. This layer may be vendor-independent. In Figure 18 we indicate a vendor-independent `hostsetting.yang` model which contains a unified model for host settings like DNS and NTP.

4. *Implementation Dependent Instances* - The concrete device configuration in the NCS Device Manager. This is the actual configuration that is sent to the devices in order to achieve the service instantiation. In the specific example of a web site it is the configuration change to the load balancers and web servers.

5. *Configuration Files* - The NETCONF XML, `editconfig`, payload sent to the devices. Note however whereas most tools work with configuration files, NETCONF does fine-grained configuration changes.

6. *Bit-Configurations* - Disk images are not directly managed by NETCONF as such.

When it comes to the specification language we have a uniform approach based on YANG at all lev-

els. Delaet and Joosen characterize the specification language from four perspectives: language-based or user-interface-based, domain coverage, grouping mechanism and multi-level specification. We will elaborate on these perspectives below.

We certainly focus on a *language-based approach* which can render various interface representations. Users can edit the configuration using the auto-rendered CLI and Web UI. You can also feed NCS with the NETCONF XML encoding of the YANG models. NCS is a *general purpose* solution in that the domain is defined by the YANG models and not the system itself. YANG supports *groupings* at the modeling level and NCS supports groupings of instance configurations as configurable templates. Templates can be applied to groups of devices.

NCS supports multi-level specifications which in Delaets and Joosens taxonomy refers to the ability to transform the configuration specifications to other formats. In our case, we are actually able to render Cisco CLI commands automatically from the configuration change. This is a topic of its own and will not be fully covered here. However NCS supports YANG model-driven CLI engines that can be fed with a YANG data-model and the engine is capable of rendering the corresponding CLI commands.

Consistency has three perspectives in the taxonomy: dependency modeling, conflict management, and workflow management. We do not cover workflow management. We consider workflow systems to be a client to NCS. NCS manages dependencies and conflicts based on constraints in the models and runtime policies. The model constraints specifiy dependencies and rules that are constrained by the model itself while policies are runtime constrained defined by system administrators. We use XPATH [6] expressions in both contexts.

Regarding conflict management NCS will detect conflicts as violations to policies as described above. The result is an error message when the user tries to commit the conflicting configuration.

The final component of the taxonomy covers the aspect of *distribution*. NCS supports a fine-grained AAA system that lets different users and client systems perform different tasks. The agent is a NETCONF client on the managed devices. The NCS server itself is centralized. The primary reason here is to enable quick validation of cross-device policy validation. The performance is guaranteed by the in-memory database.

5.2 Comparison to other major configuration management tools

There are many well-designed configuration management tools like: CFengine [5], Puppet [15], LCFG [2] and Bcfg2 [8]. These tools are more focused on system and host configuration whereas we focus mostly on network devices and network services. This is mostly determined by the overall approach taken for configuration management. In our model the management system has a data-model that represents the device and service configuration. Administrators and client programs express an imperative desired change based on the data-model. NCS manages the overall transaction by the concept of a candidate and running database which is a well-established principle for network devices.

Many host-management uses concepts of centralized versioned configuration files rather than a database with roll-back files. Also in a host environment you can put your specific agents on the hosts which is not the case for network devices. Therefore a protocol based approach like NETCONF/YANG is needed.

Another difference is the concept of desired state. For host configuration it is important to make sure that the hosts follow a centrally defined configuration which is fairly long-lived. In our case we are more focused on doing fine-grained real-time changes based on requirements for new services. There is room for combination of the two approaches where host-based approaches focused on configuration files address the more static setup of the device and our approach on top if that addresses dynamic changes.

It is also worth-while noting that most of the existing tools have made up their own specific languages to describe configuration. YANG is a viable options for the above mentioned tools to change to a standardized language.

There is of course a whole range of commercial tools, like Telcordia Activator [21], HP Service Activator [12], Amdocs [1], that address network and service configuration. While they are successfully being used for service configuration, the underlying challenges of cost and release-cycles for device adapters and flexibility of service models can be a challenge.

6 Conclusion and Future Work

6.1 Conclusion

We have shown that a standards-based approach for network configuration based on NETCONF and YANG can ease the configuration management scenarios for operators. Also the richness of YANG as a configuration description language lends itself to automating not only the device communication but also the rendering of interfaces like Command Line Interfaces and Web User Interfaces. Much of the value in this IETF standard lies in the transaction-based approach to configuration management and a rich domain-specific language to describe the

configuration and operational data. We used Erlang and in-memory database technology for our reference implementation. These two choices provide performance for parallel configuration requests and fast validation of configuration constraints.

6.2 Future Work

We have started to work on a NETCONF SNMP adaptation solution which is critical to migrate from current implementations. This will allow for two scenarios: read-only and read-write. The read-only view is a direct mapping of SNMP MIBs to corresponding NETCONF/YANG view, this mapping is being standardized by IETF [13]. The read-write view is more complex and cannot be fully automated. The main reason is that the transactional capabilities and dependencies between MIB variables are not formally defined in the SNMP SMI, for example it is common that you need to set one variable before changing others. We are working on catching the most common scenarios and define YANG extensions for those in order to automatically render as much as possible.

Furthermore we are working on a solution where we can have hierarchical NCS systems in order to cover huge networks like nation-wide Radio Access Networks. We will base this on partitioning of the instantiated model into separate CDBs. NCS will then proxy any NETCONF requests to the corresponding NCS system.

We are also working on two interesting features in order to understand the service configuration versus the device configuration: "dry-run" and "service check-sync". Committing a service activation request with dry-run calculates the resulting configuration changes to the devices and displays the diff without committing it. This is helpful in a what-if scenario: "If I provision this VPN, what happens to my devices?". The service check-sync feature will compare a service instance with the actual configuration that is on devices and display any conflicting configurations. This is useful to detect and analyze if and how the device configurations have been changed by any local tools in a way that breaks the service configurations.

References

[1] AMDOCS. Amdocs service fulfillment, 2011. http://www.amdocs.com/Products/OSS/Pages/Service-Fulfillment.aspx.

[2] ANDERSON, P., SCOBIE, A., ET AL. Lcfg: The next generation. In *UKUUG Winter conference* (2002), Citeseer.

[3] ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. Concurrent Programming in ERLANG, 1993.

[4] BJORKLUND, M. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020 (Proposed Standard), Oct. 2010.

[5] BURGESS, M. A tiny overview of cfengine: Convergent maintenance agent. In *Proceedings of the 1st International Workshop on Multi-Agent and Robotic Systems, MARS/ICINCO* (2005), Citeseer.

[6] CLARK, J., DEROSE, S., ET AL. XML path language (XPath) version 1.0. *W3C recommendation* (1999).

[7] DELAET, T., AND JOOSEN, W. Survey of configuration management tools. *Katholieke Universiteit Leuven, Tech. Rep* (2007).

[8] DESAI, N., LUSK, A., BRADSHAW, R., AND EVARD, R. Bcfg: A configuration management tool for heterogeneous environments. *Cluster Computing, IEEE International Conference on 0* (2003), 500.

[9] ENCK, W., MCDANIEL, P., SEN, S., SEBOS, P., SPOEREL, S., GREENBERG, A., RAO, S., AND AIELLO, W. Configuration management at massive scale: System design and experience. In *Proc. of the 2007 USENIX: 21st Large Installation System Administration Conference (LISA '07)* (2007), pp. 73–86.

[10] ENNS, R. NETCONF Configuration Protocol. RFC 4741 (Proposed Standard), Dec. 2006.

[11] ERLANG.ORG. The erlang programming langugage, 2011. http://www.erlang.org/.

[12] HP. HP Service Activator, 2011. http://h20208.www2.hp.com/cms/solutions/ngoss/fulfillment/hpsa-suite/index.html.

[13] J. SCHÖNWÄLDER. Translation of SMIv2 MIB Modules to YANG Modules. Internet-Draft, July 2011. http://tools.ietf.org/html/draft-ietf-netmod-smi-yang-01.

[14] JUNIPER. Junos XML Management Protocol, 2011. http://www.juniper.net/support/products/junoscript/.

[15] KANIES, L. Puppet: Next-generation configuration management.; login: the USENIX Association newsletter, 31 (1), 2006.

[16] MOBERG, C. A 30 Minute Introduction To NETCONF and YANG, 2011. http://www.slideshare.net/cmoberg/a-30minute-introduction-to-netconf-and-yang.

[17] PUGH, W. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM 33* (June 1990), 668–676.

[18] SCHOENWAELDER, J. Overview of the 2002 IAB Network Management Workshop. RFC 3535 (Informational), May 2003.

[19] SCHÖNWÄLDER, J., BJÖRKLUND, M., AND SHAFER, P. Network configuration management using NETCONF and YANG. *Communications Magazine, IEEE 48*, 9 (sept. 2010), 166 –173.

[20] SCOTT, M., AND BJORKLUND, M. YANG Module for NETCONF Monitoring. RFC 6022 (Proposed Standard), Oct. 2010.

[21] TELCORDIA. Telcordia activator, 2011. http://www.telcordia.com/products/activator/index.html.

[22] TORSTENDAHL, S. Open telecom platform. *Ericsson Review(English Edition) 74*, 1 (1997), 14–23.

[23] TRAN, H., TUMAR, I., AND SCHÖNWÄLDER, J. Netconf interoperability testing. In *Scalability of Networks and Services*, R. Sadre and A. Pras, Eds., vol. 5637 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 83–94. 10.1007/978-3-642-02627-0-7.

[24] XU, H., AND XIAO, D. Considerations on NETCONF-Based Data Modeling. In *Proceedings of the 11th Asia-Pacific Symposium on Network Operations and Management: Challenges for Next Generation Network Operations and Service Management* (2008), Springer, p. 176.

[25] YU, J., AND AL AJARMEH, I. An Empirical Study of the NETCONF Protocol. In *Networking and Services (ICNS), 2010 Sixth International Conference on* (march 2010), pp. 253 –258.

Deploying IPv6 in the Google Enterprise Network. Lessons learned.

Haythum Babiker <*haythum@google.com*>
Irena Nikolova <*iren@google.com*>
Kiran Kumar Chittimaneni <*kk@google.com*>

Abstract

This paper describes how we deployed IPv6 in our corporate network in a relatively short time with a small core team that carried most of the work, the challenges we faced during the different implementation phases, and the network design used for IPv6 connectivity.

The scope of this document is the Google enterprise network. That is, the internal corporate network that involves desktops, offices and so on. It is not the network or machines used to provide search and other Google public services.

Our enterprise network consists of heterogeneous vendors, equipment, devices, and hundreds of in-house developed applications and setups; not only different OSes like Linux, Mac OS X, and Microsoft Windows, but also different networking vendors and device models including Cisco, Juniper, Aruba, and Silverpeak. These devices are deployed globally in hundreds of offices, corporate data centers and other locations around the world. They support tens of thousands of employees, using a variety of network topologies and access mechanisms to provide connectivity.

Tags: IPv6, deployment, enterprise, early adoption, case study.

1. Introduction

The need to move to IPv6 is well-documented and well-known - the most obvious motivation being IANA IPv4 exhaustion in Feb 2011. Compared to alternatives like Carrier-Grade NAT, IPv6 is the only strategy that makes sense for the long term since only IPv6 can assure the continuous growth of the Internet, improved openness, and the simplicity and innovation that comes with end-to-end connectivity.

There were also a number of internal factors that helped motivate the design and implementation process. The most important was to break the chicken-or-egg problem, both internally and as an industry. Historically, different sectors of the Internet have pointed the finger at other sectors for the lack of IPv6 demand, either for not delivering IPv6 access to users to motivate content or not delivering IPv6 content to motivate the migration of user networks. To help end this public stalemate, we knew we had to enable IPv6 access to Google engineers to launch IPv6-ready products and services.

Google has always had a strong culture of innovation and we strongly believed that IPv6 will allow us to build for the future. And when it comes to universal access to information we want to provide it to all users, regardless of whether they connect using IPv4 or IPv6.

We needed to innovate and act promptly. We knew that the sooner we started working with networking equipment vendors and with our transit service providers to improve the new protocol support, the earlier we could adopt the new technology and shake the bugs out. Another interesting problem we were trying to solve in our enterprise organization was the fact that we are running tight on private RFC1918 addresses - we wanted to evaluate techniques like Dual-Stack Lite, i.e to make hosts IPv6-only and run DS-Lite on the hosts to provide IPv4 connectivity to the rest of the world if needed.

2. Methodology

Our project started as a grass-roots activity undertaken by enthusiastic volunteers who followed the Google practice of contributing 20% of their time to internal projects that fascinate them. The first volunteers had to learn about the new protocol from books and then plan labs to start building practical experience. Our essential first step was to enable IPv6 on our corporate network, so that internal services and applications could follow.

Our methodology was driven by four principles:

1. Think globally and try to enable IPv6 everywhere: in every office, on every host and every service and application we run or use inside our corporate network.
2. Work iteratively: plan, implement, and iterate launching small pieces rather than try to complete everything at once.
3. Implement reliably: Every IPv6 implementation had to be as reliable and capable as the IPv4 ones, or else no one would use and rely on the new protocol connectivity.
4. Don't add downtime: Fold the IPv6 deployments into our normal upgrade cycles, to avoid additional network outages.

3. Planning and early deployment phases

First, we started creating a comprehensive addressing plan for the different sized offices, campus buildings, and data centers. Our initial IPv6 addressing scheme followed the guidelines specified in RFC5375 (IPv6 Unicast address assignment):

- Assign /64 for each VLAN
- Assign /56 for each building
- Assign /48 for each campus or office

We decided to use the Stateless Address Auto-Configuration capability (SLAAC) for IPv6 address assignments to end hosts. This stateless mechanism allows a host to generate its own addresses using a combination of locally available information and information advertised by routers, thus no manual address assignment is required.

As manually configuring IP addresses has never really been an option, this approach addressed various operating systems DHCPv6 client support limitations and therefore sped the rollout of IPv6. It also provides a seamless method to re-number and provide address privacy via the privacy extension feature (RFC 4941). Meanwhile, we also requested various sized IPv6 space assignments from the Regional Internet Registries. Having PI (Provider Independent) IPv6 space was required to solve any potential multihoming issues with our multiple service providers.

Next, we had to design the IPv6 network connectivity itself. We obviously had several choices here; we preferred dual-stack if possible, but if not then we had to build different types of tunnels (as a 6-to-4 transitioning mechanism) on top of the existing IPv4 infrastructure or to create a separate IPv6 infrastructure. The latter was not our preferred choice since this would have meant the need for additional time and resources to order data circuits and to build a separate infrastructure for IPv6 connectivity.

We also tried to design a scalable IPv6 backbone to accommodate all existing WAN clouds (MPLS, Internet Transit and the Google Production network, which we use as our service provider for some of the locations). Along with the decision to build the IPv6 network on top of the existing physical one we tried to keep the IPv6 network design as close to the IPv4 network in terms of routing and traffic flows as possible. The principle of changing only the minimum amount necessary was applied here.

By keeping the IPv6 design simple, we wanted to ensure scalability and manageability; also it is much easier for the network operations team to support it. In order to comply with this policy we decided to use the following routing protocols and policies:

- HSRPv2 - First hop redundancy
- OSPFv3 - Interior gateway protocol
- MP-BGP - Exterior gateway protocol
- SLAAC - for IP addresses assignments for the end hosts.

Our proposed routing policy consist of the following rules: we advertise the office aggregate routes to the providers, while only accept the default route from the transit provider.

We also aggressively started testing and certifying code for the various hardware vendors' platforms and working on building or deploying IPv6 support into our in-house built network management tools.

In 2008 we got our first ARIN-assigned /40 IPv6 space for GOOGLE IT and we deployed a single test router having a dual-stacked link with our upstream transit provider. The reason for having a separate device was to be able to experiment with non-standard IOS versions and also to avoid the danger of having higher resource usage (like CPU power).

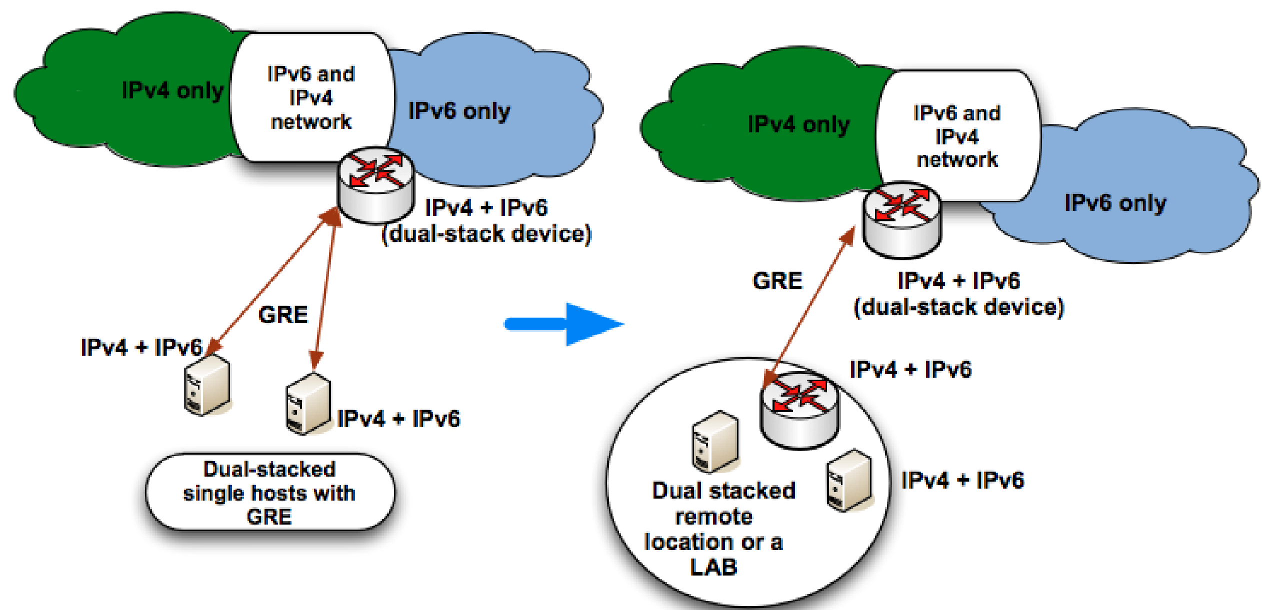


Figure 1: phase I - dual-stack separate hosts and labs

The early enthusiasts and volunteers to test the IPv6 protocol had one GRE tunnel each running from their workstations to this only IPv6 capable router, which was sometimes giving around 200ms latency, due to reaching relatively closely located IPv6 sites via a bro-

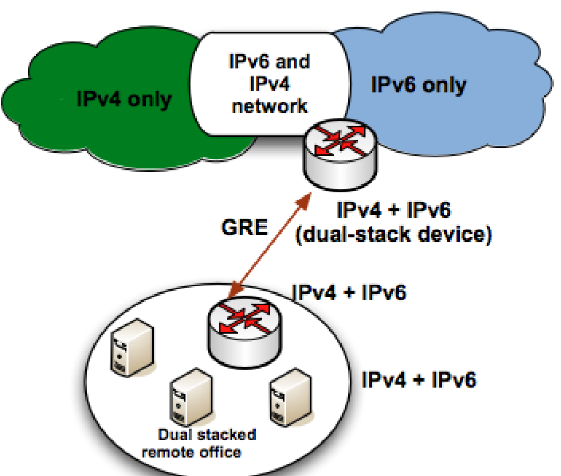


Figure 2: phase II - dual-stack offices

ker device on the other side of the world.

The next steps during this initial implementation phase were to create several fully dual-stacked labs (Figure 1) and connect them to the dual-stacked router using the same GRE tunnels, but instead of at certain hosts, these GRE tunnels were now terminated at the lab routers. In the next phase we started dual-stacking entire offices and campus buildings (Figure 2) and then building a GRE tunnel from the WAN Border router at each location to the egress IPv6 peering router.

In the third phase we started dual-stacking entire offices, while trying to prioritize deployment in offices with immediate need for IPv6 (Figure 3), e.g. engineers working on developing or supporting applications for IPv6.

Using this phased approach allowed us to gradually gain skills and confidence and also to confirm that IPv6 is stable and manageable enough to be deployed in our network globally.

4. Challenges

We faced numerous challenges during the planning and deployment phases, not only technical, but also administrative and organizational such as resource assignment, project prioritization and the most important - education, training and gaining experience.

4.1 Networking challenges

The most important technical issue we faced was the fact that the major networking vendors lack enterprise IPv6 features, especially on some of the mid-range devices and platforms. Also certain hardware platforms support IPv6 in software only, which causes high CPU usage when the packets are handled by the software. This has a severe performance impact when using access control lists (ACLs). In another example of limitations with some of our routing platfors vendors, the only IPv6 tunneling mechanism available is Generic Routing Encapsulation (GRE). The main reason for this partial IPv6 implementation in the networking devices is that most vendors are not even running IPv6 in their own

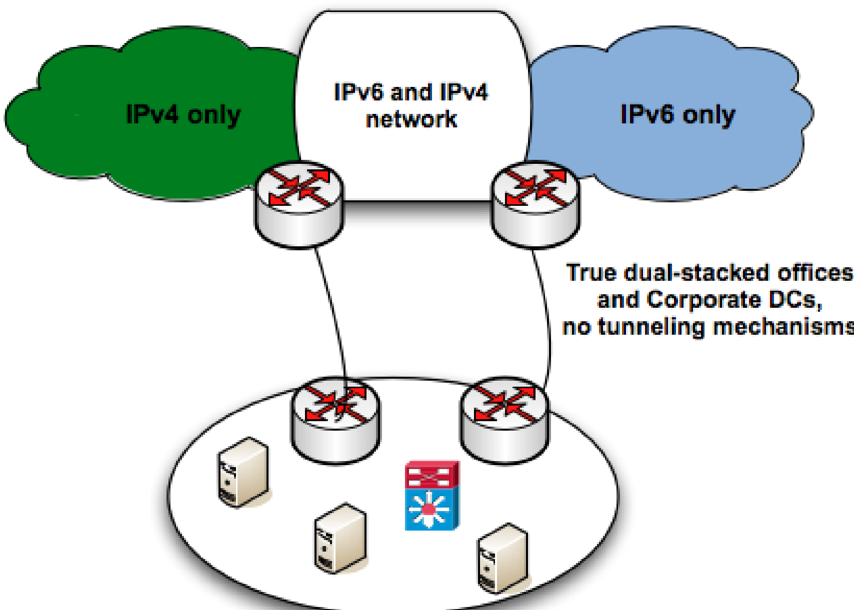


Figure 3: phase III - dual-stack the upstream WAN connections to the transit and MPLS VPN providers

corporate networks. Also the TCAM table in one of the switch platforms we use is limited when you enable an IPv6 SDM routing template. Another example of a network challenge is the software only routing support of IPv6 in the platforms we deploy as wireless core switches.

Our wireless equipment vendor did not have support for IPv6 ACLs and currently lacks support for IPv6 routing. We also faced the problem with VLAN pooling on the wireless controllers - in that mechanism, the wireless controller assigns IP addresses from the different VLANs (subnets) on a round-robin basis as each wireless client logs in. We wanted to utilize multiple VLANs using this technique to provide easy address management and scalability. However, the VLANs pooling implementation on our specific vendor leaked IPv6 neighbor discovery and multicast Router Advertisements (RAs) between the VLANs. This introduced IPv6 connectivity issues as the clients were able to see multiple RAs from outside the client VLANs. The solution provided by the vendor in a later software release was to implement IPv6 firewalling to restrict the neighbor discovery and Routers Announcement multicast traffic leaking across VLANs.

One more example is the WAN Acceleration devices we use in our corporate network - we cannot encrypt or accelerate IPv6 traffic using WCCP (Web Cache Control Protocol), since the current protocol standard (WCCPv2) does not even support IPv6 and thus is not implemented on the devices. Currently we are evaluating workarounds like PBR (Policy Based Routing) to overcome this

with the dual-stack infrastructure is getting a feel how much traffic on the links is IPv4 and how much IPv6. We still needed to work on collecting, parsing, and properly displaying Netflow stats for IPv6 traffic. The problem that we have here is due to a specific routing platform vendor that is no longer developing the OS branch for the specific hardware model we use, while the current OS versions do not support NetFlow v9.

We also faced some big challenges when working with various service providers. The SLA that they support is very different than the SLA for IPv4, and, in our experience, the implementation time for turning up IPv6 peering sessions takes much longer than IPv4 ones. In addition, our internal network monitoring tools were unable to alert on base monitoring for IPv6 connectivity until recently.

4.2 Application and client software

The main problem was that the many application whitelists we use for multiple internal applications were initially not developed to support IPv6, so when we first started implementing IPv6 the users on the IPv6 enabled VLANs and offices were not able to reach lots of our internal online tools. We even got some false positive security reports saying that some unknown addresses were trying to access restricted online applications.

In order to fight this problem, we aimed at phasing out old end-host OSes and applications that do not support IPv6 or where IPv6 is disabled by default. Although we no longer support obsolete host OSes in our corporate network, there are still some IPv6 related issues with

connectivity might be broken due to problems with the remote ISATAP router and infrastructure.

We also still have not fully solved the printer problem, an most do not support IPv6 at all or just for management.

Unfortunately large groups of systems and applications exist that cannot be easily modified, even to enable IPv6 - for example heavy databases and some of the billing applications due to the critical service they offer. And on top of that, the systems administrators are often too busy with other priorities and do not have the cycles to work on IPv6 related problems.

5. Lessons Learned

We learned a lot of valuable lessons during the deployment process. Unfortunately, the majority of the problems we've faced were unexpected.

Since lots of providers still do not offer dual-stack support to the CPE (customer-premises equipment), we had to use manually built GRE over IPSec tunnels to provide IPv6 connectivity for our distributed offices and locations.

Creating tunnels causes changes in the maximum transmission unit (MTU) of the packets. This often causes extra load on the router's CPU and memory, and all possible fragmentation and reassembly adds extra latency. Since we often do not have full control over the network connectivity from end to end (e.g. between the different office locations) we had to lower the IPv6 path MTU to 1416 to avoid possible packets being lost due to lost ICMPv6 messages on the way to the destination.

Another big problem we had to deal with was the end host OSes immature IPv6 support. For example, some of them still prefer IPv4 over IPv6 connectivity by default. Some others do not even have IPv6 connectivity turned on by default, which makes the users of this OS incapable of testing and providing feedback for the IPv6 deployment. It also turned out that another popular host OS does not have client support for DHCPv6 and thus we were forced to go with SLAAC for assigning IPv6 addresses to the end hosts.

We ran into countless applications problems too: No WCCP support for IPv6, no proxy, no VoIP call managers, and many more. When trying to talk to the vendors they were always saying - *if there is a demand for IPv6 support at all, we've never heard it before.*

In summary, when it comes to technical problems we can confirm that there is a lot of new, unproven and therefore buggy code, and getting our vendors aligned so that everything supports IPv6 has been a challenge.

Regarding the organizational lessons we learned - the most important one is that IPv6 migration potentially touches everything, and so migrating just the network or just a single service or application or platform does not make sense by itself. This project also turned out to be a much longer term project than originally intended. We've been working on this project for 4 years already and we are still probably only half way to completion. Still, the biggest challenge is not deploying IPv6 itself, but integrating the new protocol in all management procedures and applying all IPv4 current practice concepts for it too - for example the demand for redundancy, reliability and security.

6. Summary

The migration to IPv6 is not an L3 problem. It is more of an L7-9 problem: resources, vendor relationship/management, and organizational buy-in. The networking vendors' implementations mostly work, but they do have bugs: we should not expect something to work just because it is declared supported.

Because of that we had to test every single IPv6 related feature, then if a bug was found in the lab we reported it and kept on testing!

7. Current status and future work

Around 95% of the engineers accessing our corporate network have IPv6 access on their desks and are whitelisted for accessing Google public services (search, Gmail, Youtube etc.) over IPv6. This way they can work on creating, testing and improving IPv6 aware applications and Google products. At the same time internally we keep on working on enabling IPv6 support on all our internal tools and applications used in the corporate network.

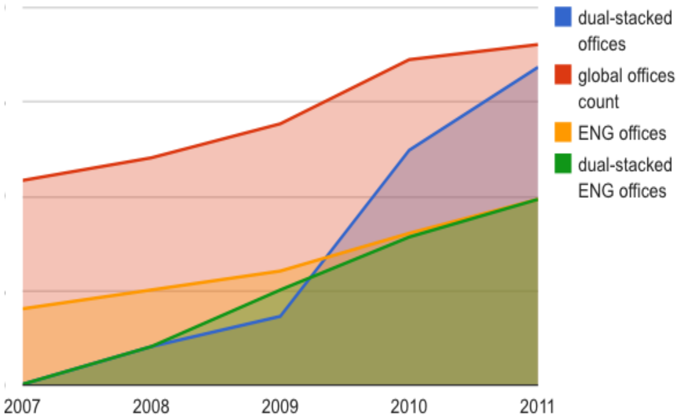


Figure 4: Timeline for dual-stacking Google corporate locations

In the long run, the potential of introducing DHCPv6 (state-full auto-configuration) can be investigated given the advantages of DHCP flexibility and better management. However enabling this functionality still depends on the support of the end hosts DHCPv6 client on the desktop platforms.

We also want to revisit the IP addressing allocation of /64 to every subnet on the corporate network, since a new RFC 6164 has been published that recommends assigning /127 addresses on P2P links.

Since the highest priority for all organizations is to IPv6-enable their public-facing services, following our experience we can confirm - dual-stack works well today as a transition mechanism!

There is still quite a lot of work before IPv4 can be turned off anywhere, but we are working hard towards it. The ultimate goal is to successfully support employees working on an IPv6-only network.

Experiences with BOWL: Managing an Outdoor WiFi Network (or How to Keep Both Internet Users and Researchers Happy?)

T. Fischer, T. Hühn, R. Kuck, R. Merz, J. Schulz-Zander, C. Sengul
TU Berlin/Deutsche Telekom Laboratories
{thorsten,thomas,rkuck,julius,cigdem}@net.t-labs.tu-berlin.de, ruben.merz@telekom.de

Abstract

The Berlin Open Wireless Lab (BOWL) project at Technische Universität Berlin (TUB) maintains an outdoor WiFi network which is used both for Internet access and as a testbed for wireless research. From the very beginning of the BOWL project, we experienced several development and operations challenges to keep Internet users and researchers happy. Development challenges included allowing multiple researchers with very different requirements to run experiments in the network while maintaining reliable Internet access. On the operations side, one of the recent issues we faced was authentication of users from different domains, which required us to integrate with various external authentication services. In this paper, we present our experience in handling these challenges on both development and operations sides and the lessons we learned.
Keywords: WiFi, configuration management, authentication, research, DevOps, infrastructure, testbed

1 Introduction

Wireless testbeds are invaluable for researchers to test their solutions under real system and network conditions. However, these testbeds typically remain experimental and are not designed for providing Internet access to users. In the BOWL project [2, 7, 9], we stepped away from the typical and designed, deployed and currently maintain a live outdoor wireless network that serves both purposes. The benefits are twofold [9]:

- University staff and students have outdoor wireless network access. Our network covers almost the entire Technische Universität Berlin (TUB) campus in central Berlin (see Fig. 1).
- Researchers have a fully reconfigurable research platform for wireless networking experimentation that includes real network traffic (compared to synthetic traffic).

During its lifetime, the BOWL network has significantly evolved from a prototype architecture and design in 2009 [7, 9] towards a production network, which brings out several administrational and development challenges. The network and its components, including traffic generators, routers and switches interconnect with

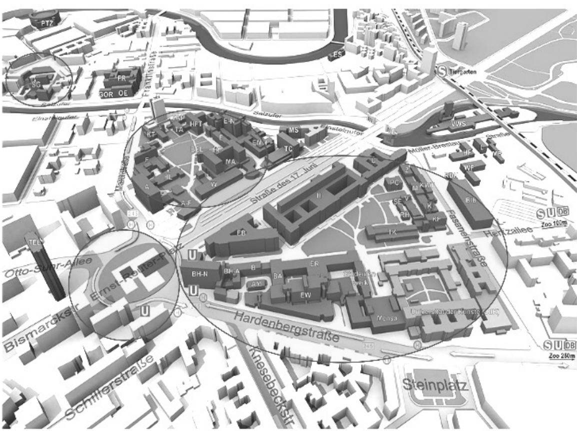


Figure 1: Coverage of the BOWL network on the TU-Berlin campus.

a variety of other networks and infrastructures which are not controlled by the BOWL project, adding to the inherent complexity of running a production network. In this paper, we focus on two of our many challenges that we have experienced in the last year while moving from a prototype to a more stable infrastructure. We present our challenges from the perspective of development and network operations and its reliance on external services, respectively.

Development challenges were - and still are - numerous [9]. The most prominent is the variety of people that work on different subsets of network components, and change network configuration and operating system images. The requirements for associated services and infrastructure, as well as the research goals, continuously change as we and other users change the way the BOWL network is used on a daily basis. In fact, our experience showed that it was necessary to rewrite the BOWL software significantly during the development as well as the operational lifetime of the BOWL project. Many of the changes were also triggered with the feedback received from external users.

From a purely operational point of view, authentication of users to the BOWL network has proven surprisingly complex. A project-specific remote authentication dial-in user service (RADIUS) installation is used as the

pivot point to integrate a number of other distributed and disparate authentication solutions. Users include (1) centrally managed university IT accounts, (2) users from our own department, (3) users of the affiliated external institution Deutsche Telekom Laboratories (hereafter T-Labs), (4) project-only user accounts, and (5) eduroam users. TUB user authentication is a critical part of the contractual relationship with the university central IT department. The major challenge we faced and still face is the recovering from errors that might lie in external authentication services that we rely on to support these accounts. In this paper, we present a major outage we went through due to such problems and the lessons we learned.

2 BOWL (Berlin Open Wireless Lab) and DevOps Challenges

The main task of the BOWL project is to satisfy two somehow conflicting requirements from two user groups – Internet users and researchers (which are often developers). We see the following requirements as DevOps challenges:

- **Researchers demand a configurable network (development):** The testbed is intended for a wide selection of research topics ranging from enhancing measurement-based physical layer models for wireless simulation [8] to routing protocols [11, 12]. Hence, one of the goals of the BOWL project is to allow multiple researchers to access the network, deploy experimental services, change configurations and run new experiments or repeat old experiments while still ensuring Internet access. Therefore, the BOWL project required the development of several tools to automate software and configuration deployment in the testbed. We discuss our experience with these tools, and how they evolved in Section 4.
- **Internet users demand a reliable network (operation):** Changing the network configuration, deploying and running experiments should not affect the availability of Internet access. This implies that basic connectivity should not be affected, or only for a negligible time duration. It also means that services such as authentication, DHCP and DNS need to remain available in any experiment setup. How BOWL network architecture addressed this problem is summarized in Section 3. A major operational challenge is the authentication of different type of users (e.g., Internet users from TUB and T-Labs, and researchers) to the BOWL network, and we discuss this in detail in Section 5.

3 BOWL Network Architecture

In addition to its outdoor network, the BOWL project is in charge of two additional networks: (1) a smoketest network, for early development and testing and (2) an indoor network, for small-scale deployment and testing. These networks are used for development and staging before a full-scale deployment and measurements in the outdoor network. Therefore, the research usage pattern of the outdoor network is more bursty, with periods of heavy activity followed by lighter usage, whereas the smoketest and the indoor networks have been in heavy use since their deployment in early 2008. In this paper, we mainly focus on our experience with the outdoor network.

The BOWL network architecture was first presented in [9]. In this section, we summarize this architecture to give the necessary information to understand the BOWL environment and its challenges. The outdoor network comprises more than 60 nodes deployed on the rooftops of TUB buildings. It spans three different hardware architectures (ARM, MIPS and x86). Each node is powered by Power over Ethernet (PoE), which simplifies cabling requirements. All nodes are equipped with a hardware watchdog, multiple IEEE 802.11a/b/g/n radio interfaces and a wired Ethernet interface. One radio interface is always dedicated to Internet access, the additional radio interfaces are free to be used in research experiments, and the wired interface is used for network management and Internet connectivity. All nodes are connected via at least 100 Mbit/s Ethernet to a router that is managed by the project. A VLAN network ensures a flat layer 2 connectivity from our router to each node. Our router ensures connectivity to the BOWL internal network, the TUB network and the Internet. In its default configuration (which is called the *rescue* configuration), the network is set up as a bridged layer 2 infrastructure network. Association to the access interface and encryption of the traffic is protected by WPA2 (from the standard IEEE 802.11i [1]). Authentication is performed with IEEE 802.1x and RADIUS.

Each node runs OpenWrt [5] as the operating system. The OpenWrt build system typically produces a minimally configured image. To tailor this image to each node, the image is configured at boot time by an auto-configuration system that applies a so-called *configuration* to the image. A configuration includes all the configuration files that go under the `/etc/config` directory (the layout is specific to OpenWrt), and additional files, scripts and packages that may be needed by the experimenter. The details of the auto-configuration system are explained in Section 4.

By default, every node runs a default *rescue* image and uses the aforementioned *rescue* configuration. Researchers install *guest* images in extra partitions and

use *guest* configurations. Because of the unique needs of experiment monitoring and reconfiguration at runtime, a network management and experiment monitoring system was developed, which also went through significant changes from its version presented in [9]. In essence, it comprises two main components: a node-controller, which runs on each node and a central node-manager. Each node-controller connects to one node-manager. However, with the recent changes, several node-managers can be now run in parallel i.e. one for each experiment if several parallel experiments are needed to be run or for development. Our typical operation requires one node manager per network (e.g., smoketest, indoor and outdoor). Thanks to the underlying VLAN infrastructure and virtualization of the central router, the traffic generated by each experiment can be isolated, if multiple experiments are running in the network. More details on this topic can be found in [9].

Unwanted side effects due to using experiment software (e.g., crashes, slowing down of network services) are expected to occur in practice but their effect needs to be minimized as much as possible. This is achieved thanks to the locally installed images. Indeed, a node that is experiencing problems can be *rescued* by an immediate reboot into the rescue image. This mode of operation is implemented making use of hardware and software watchdogs that periodically check that certain services are operational. One example is that, node-controllers at each node periodically check connectivity to the central node manager and when a disconnection is detected, the node is rebooted to the rescue image within 60s. Note that since each node independently triggers a switch to the rescue mode based on its own hardware and software watchdogs, nodes do not go down all at the same time limiting network disruptions. More details on how experiment problems are detected can be found in [9].

In the remainder of the paper, we focus on how we addressed two main challenges: the development challenge of supporting multiple network configurations for different researchers and the operational challenge of authentication in the BOWL network.

4 A Development Challenge: Supporting Multiple Network Configurations for Wireless Experimentation

One of the main goals of the BOWL project is to allow multiple researchers to create experiments, and be able to run and repeat their experiments in a consistent fashion. In the remainder of this section, we first summarize the system that we started off with around mid 2008, and describe how it evolved during the lifetime of the project. Essentially, the reliability of the

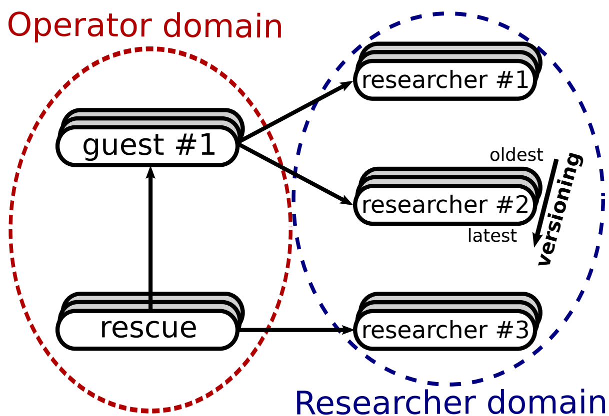


Figure 2: An example of how three researchers maintain their own configuration in the BOWL network.

BOWL network was jeopardized due to several configuration glitches and therefore, our complete software rewrite decisions were significantly affected by the need to maintain network reliability at all times.

The node configuration of a given experiment consists of two parts: (1) an operating system image and (2) an experiment configuration. OpenWrt manages the whole configuration of the operating system using the universal configuration interface (UCI)[6]. We also take advantage of the UCI. As the network is used for very different purposes, it becomes necessary to maintain consistent network configurations across the users. Therefore, initially, we had a configuration database and stand-alone scripts to apply these configurations from a central server manually. As more nodes were deployed in the BOWL network, it became a necessity to have a more scalable and manageable solution. To this end, the existing node-manager and node-controller framework was extended to support node configurations. The important components to a BOWL user are: (i) the web-based front-end to a configuration database, and (ii) a client-server auto-configuration process that runs in node-controllers and the node-manager, respectively. The auto-configuration scheme was added after mid 2010 due to the several failures that occurred with the earlier version. Figure 2 illustrates how, for instance, three researchers maintain their configurations in the BOWL system.

Using the web-based front-end, a researcher can pick a configuration, image and the node partition to deploy its experiment. From this step on, the user flashes his own image to this partition and nodes are configured by the auto-configuration process at boot time (or before the image is booted). However, currently, a researcher still needs to record the information about which image was used with which experiment configuration. In the future, we are planning to automate this lab bookkeeping

process. Finally, a reservation system prevents node and image usage conflicts. Currently, the reservation system used in BOWL is primitive, in the sense that the entire network is reserved to a single researcher for a given period of time. Each researcher is responsible of his image and configuration and deploys this image to a given node partition. Hence, merging of multiple images from different experimenters is not expected.

This framework, complete with a new auto-configuration scheme, is in use since mid 2010 by the BOWL group and visiting researchers, that also remotely access our network. We learned several lessons since then, which resulted in the current state of the framework as we use today. For instance, one issue resulted from the inheritance of configurations in the database. It was not obvious to us at the beginning that researchers would have difficulties discovering the inheritance hierarchy. But some of our early users applied changes to the base configuration expecting them to take effect in the descendant configuration. To avoid such problems, we now expose the inheritance hierarchy to the users of our system and visualize it in the web-based front-end. Finding a right way to do this also was a challenging task. Furthermore, being too accommodating was not a good idea and we ended up limiting the functionality of the web-based front-end. Earlier, researchers could push a configuration to a given node by just pressing a button. However, since installing images and configurations were separated from each other, it sometimes resulted in applying a wrong configuration to the wrong image. Therefore, we removed this functionality from the front-end. Actually, this was the main reason why an auto-configuration scheme was added to the system. A final lesson learned was not to assume any network stability during configurations. With our first auto-configuration implementation, the nodes fetched their configurations from the node-manager right after booting. However, if there were any network instabilities during this time, the watchdog would trigger and interfere with the auto-configuration. We now avoid this problem by having nodes first fetch their configurations before booting the image, configure the image, and boot only if all checks pass. While our development activities have slowed down as users become more used to working with our framework, we are still looking into simplifying things even further to lower the entry barrier of using the BOWL network.

5 An Operational Challenge: Authentication in the BOWL network

In exchange of the rooftop usage and installation support, the BOWL project has contractual obligations with TUB to provide wireless Internet access to staff and students. Hence, we need to provide the usual authentication and accounting services that would be expected from any WiFi access network. To this end, we use the widely deployed FreeRADIUS software [4], which is a server implementation of RADIUS [10]. When a user tries to authenticate to our network, the authenticator (hostapd) at the WiFi access point communicates with the RADIUS server. Using challenge-based protocols, the RADIUS server determines whether credentials provided by a user are valid. Using the results from this decision process, the access point either allows the user to join the network or rejects him.

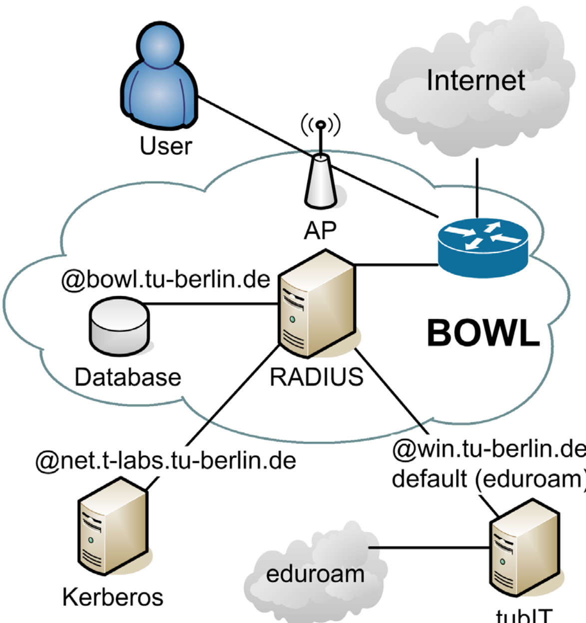


Figure 3: Logical diagram of the BOWL authentication infrastructure.

One of the main reasons that makes authentication in the BOWL network a challenging task is the interconnections with other networks and the need to provide access to different type of accounts. FreeRADIUS does support this by allowing access decisions based on local account databases or using the results of requests proxied to further upstream services, which may in turn again be other RADIUS implementations or entirely different services. Currently, the BOWL network needs to provide access for the following types of accounts (see Figure 3):

- **TUB accounts** as held by students and members of staff in another RADIUS server, administered by TUB. Access is provided using PEAP with MSCHAPv2. The BOWL network does not hold (or ever sees in any other way) passwords associated with these accounts, because it just proxies the encrypted challenge and response messages.
- **eduroam** [3] access is provided by TUB using the

same scheme as described above. Accounting data for this and the previous scheme are forwarded to TUB.

- **Accounts for the local department FG INET**, administered by the department of which BOWL is a part. The upstream authentication service is a Kerberos installation. Access is provided using TTLS with PAP, because this kind of upstream service requires that the FreeRADIUS server handles the passwords of the users.
- **Local accounts** for demonstration and guest access purposes, administered by the BOWL network. Implemented using PEAP with MSCHAPv2. Contrary to the previous schemes, all schemes available as default settings in FreeRADIUS provide working options here. The credentials are held in a local database.
- **Experiment-specific accounts** for researchers, administered by the BOWL network. Implemented in a vein similar to the local accounts. These special accounts are available for us to be able to filter out data about traffic generated for the purpose of experimentation from the accounting database.

From this list it follows immediately that support requirements towards users tend to vary with upstream authentication source. Administration and support complexity inevitably increases rapidly with additional supported schemes. This complexity which results from the highly interconnected nature of BOWL is only bound to increase. For example, there are discussions whether some parts of Deutsche Telekom Laboratories are to be provided access to BOWL using a limited subset of the accounts held in an Active Directory service. Also, there are plans to move local accounts into a LDAP installation for centralized administration.

In the process of creating all these authentication interconnections, we have learned that unlike some other pieces of server software, FreeRADIUS makes it somewhat difficult to set up a fresh installation with self-written configuration files, because of the inherent complexity of the flow of authentication requests within the server. The developers make a point of telling their users to proceed only from the default settings, making small incremental changes. Therefore, keeping the configuration files in a version control system has proven to be even more invaluable than with any other service. In summary, FreeRADIUS setup and handling can be daunting and time-consuming for the administrator who works with it extensively for the first time. However, we still feel that we have made the right choice. The software is freely available under the terms of the GPL, it works without any need for modification on the BOWL network and it provides an extremely rich feature set.

Now, monitoring of availability of external authentication services has become one of our major challenges, which requires working test accounts for those services. Monitoring software like Nagios provides support for self-written plug-ins, but not all upstream service providers are prepared to provide such accounts. Testing installations are needed, but they are hard to realize as they require testing configurations on live nodes. Furthermore, the upstream providers may be required to accept and serve requests from these testing installations. Also, obviously, it must be avoided that the accounting database is not polluted by bogus/testing data. All of this must be done carefully, as FreeRADIUS has proven to be a piece of software to which configuration changes need to be made with special care because of unintentional interactions with other configuration sections.

One important consequence from not being able to fully test and monitor external authentication services is the loss of usage of the network. This is quite annoying when it is due to problems in external services that we do not fully control. And loss of control is not just a hypothetical scenario. During the spring of 2011, no TUB users were able to authenticate to the BOWL network. Local testing revealed that the reason did not lie in the BOWL network installation; requests were passed on to the upstream server correctly. The fact that all authentication protocols in use are encrypted and stateless made further debugging difficult. The hospitalization of our main technical contact person at TUB, who was also the only person knowledgeable about the RADIUS configurations, at exactly this point in time put another obstacle in our way to successfully resolve this issue. Eventually, it was found that a server certificate of one of the upstream servers had expired, leading to rejection of user authentication attempts. Luckily, the BOWL network bounced back from this incident, and we observed a speedy uptake by users again shortly afterwards. The first power users returned the morning after the upstream servers were fixed; the number of distinct users increased continuously and two weeks later, the number of distinct users per day peaked.

The most that an operational team can do in these cases is to rely on its own monitoring tools in order to be able to find the source of problems as quickly as possible; and to build open and positive relationships with upstream operations teams that make communication and collaboration as smooth as possible. We also noticed that solving the problem was delayed due to the unavailability of the only person with the know-how. Based on this experience, on our side, we try to make sure that the BOWL system knowledge is shared among multiple people, who can handle issues independently.

6 Current State and Lessons Learned

To manage a live and experimental testbed is a significant challenge, as one needs to keep both Internet users and researchers happy. In this paper, we described the auto-configuration and authentication solutions that we run to be able to serve both communities.

We learned several lessons during this phase, which we summarize as follows:

1. It is important to have complete and thorough documentation that details the know-how of the BOWL project group. Using our system for the first time is currently not trivial. Therefore, more time needs to be invested in educating future users and simplifying operation.
2. Early adopters of the BOWL framework proved that people always find a way to use an interface differently than you expect them to. Well-defined user interfaces with less functionality turned out to be much more useful than providing more functionality with specifications unclear to the user. Therefore, it is better to design simple first, and add extra functionality when only it is absolutely required by the users.
3. While building the BOWL framework, we once more realized how important user-friendly interfaces are. People should be exposed all the necessary information to run the system correctly easily.
4. In a live network, network disruptions will happen. Therefore, all functionality should be designed around issues that can rise from network instability.
5. Our authentication problems showed that the most important thing is to maintain a good contact with all the parties that can affect operation. More than expected, the problem lies outside our own network, and we need to rely on problem solving skills of the upstream service providers.
6. FreeRADIUS configuration changes should be maintained in a version control system. This makes it a lot easier to revert to a previously working version.
7. The complexity of any important component of the network, such as authentication services, is only going to increase as the number of interconnections increases. Being aware of this fact aids in the planning of upcoming changes and aids with the integration into previously existing configuration options.
8. Finally, we learned that it is essential not to create information bottlenecks in a project team, and there should always be multiple people who know how to handle problems independently of others.

7 Acknowledgments

We thank Harald Schiöberg for the architecture of the BOWL testbeds and the implementation of the original BOWL software suite. This work was supported by Deutsche Telekom Laboratories, in the framework of the BOWL project.

References

[1] 802.11-2007 IEEE standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements - part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications.

[2] Berlin Open Wireless Lab. `http://www.bowl.tu-berlin.de/`.

[3] eduroam. `http://www.eduroam.org/`.

[4] FreeRADIUS. `http://www.freeradius.org/`.

[5] OpenWrt. `http://openwrt.org/`.

[6] The UCI System. `http://wiki.openwrt.org/doc/uci`.

[7] M. Al-Bado, A. Feldmann, T. Fischer, T. Hühn, R. Merz, H. Schiöberg, J. Schulz-Zander, C. Sengul, and B. Vahl. Automated online reconfigurations in an outdoor live wireless mesh network. In *Proceedings of the ACM SIGCOMM Conference (demo session)*, August 2009.

[8] M. Al-Bado, R. Merz, C. Sengul, and A. Feldmann. A site-specific indoor link model for realistic wireless network simulations. In *4th International Conference on Simulation Tools and Techniques (SimuTools)*, 2011.

[9] R. Merz, H. Schiöberg, and C. Sengul. Design of a configurable wireless network testbed with live traffic. In *Proceedings of TridentCom 2010*, volume 46 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (LNICST)*, pages 189–198. Springer, May 2010.

[10] C. Rigney, S. Willens, A. Rubens, and W. Simpson. Remote Authentication Dial In User Service (RADIUS), 2000.

[11] N. Sarrar. Implementation and evaluation of an opportunistic mesh routing protocol. Master's thesis, Technische Universität Berlin, 2009.

[12] F. Sesser. A performance analysis of scalable source routing (ssr) in real-world wireless networks. Master's thesis, Technische Universität München, 2011.

Why Do Migrations Fail and What Can We Do about It?

Gong Zhang and Ling Liu
College of Computing, Georgia Institute of Technology, Atlanta, USA

Abstract

This paper investigates the main causes that make the application migration to Cloud complicated and error-prone through two case studies. We first discuss the typical configuration errors in each migration case study based on our error categorization model, which classifies the configuration errors into seven categories. Then we describe the common installation errors across both case studies. By analyzing operator errors in our case studies for migrating applications to cloud, we present the design of CloudMig, a semi-automated migration validation system with two unique characteristics. First, we develop a continual query (CQ) based configuration policy checking system, which facilitate operators to weave important configuration constraints into CQ-based policies and periodically run these policies to monitor the configuration changes and detect and alert the possible configuration constraints violations. Second, CloudMig combines the CQ based policy checking with the template based installation automation to help operators reduce the installation errors and increase the correctness assurance of application migration. Our experiments show that CloudMig can effectively detect a majority of the configuration errors in the migration process.
Keywords*:System management, Cloud Computing, Application Migration*
Technical area*: Cloud Computing*

1 Introduction

Cloud computing infrastructures, such as Amazon EC2 [3], provide elastic, economical and scalable solutions and outsourcing opportunities for different types of consumers and end-users. Its pay-as-you-go utility-based computing model attracts many enterprises to build their information technology services and applications on the EC2-like cloud platform(s) and many successfully achieve their business objectives, such as SmugMug, Twistage and so forth. An increasing number of enterprises embrace Cloud computing by making their deployment plans or engaging in the process to migrate their services or applications from a local data center to the Cloud computing platform like EC2, because this will greatly reduce their infrastructure investments, simplify operations, and obtain better quality of information service.

However, the application migration process from the local data center to the Cloud environment turns out to be quite complicated: error-prone, time-consuming and costly. Even worse, the application may not work correctly after the sophisticated migration process. Existing approaches mainly complete this process in an ad-hoc manual manner and thus the chances of error are very high. Thus how to migrate the applications to the Cloud platform correctly and effectively poses a critical challenge for both the research community and the computing service industry.

In this paper, we investigate the factors and the causes that make the application migration process complicated and error-prone through two case studies, which migrate Hadoop distributed system and RUBiUS multi-tier Internet service from a local data center to Amazon EC2. We first discuss the typical configuration errors in each migration case study based on our error categorization model, which classifies the configuration errors into seven categories. Then we describe the common installation errors across both case studies. We illustrate each category of errors by examples through selecting a subset of the typical errors observed in our experiments. We also present the statistical results on the error distributions in each case study and across case studies. By analyzing operator errors in our case studies for migrating applications to cloud, we present the design of CloudMig, a semi-automated migration validation system that offers effective configuration management to simplify and facilitate the migration configuration process. The CloudMig system makes two unique contributions. First, we develop a continual query based configuration policy checking system, which facilitate operators to weave important configuration constraints into continual query policies and periodically run these policies to monitor the configuration changes

and detect and alert the possible configuration constraints violations. Second, CloudMig combines the continual query based policy checking system with the template based installation automation system, offering effective ways to help operators reduce the installation errors and increase the correctness assurance of application migration. Our experiments show that CloudMig can effectively detect a majority of the configuration errors in the migration process.

In the following sections, we discuss the potential causes that lead to the complicated and error-prone nature of the migration process in Section 2. We review the existing approaches and their limitations in Section 3. We report our operator-based case studies in Section 4 through a series of experiments conducted on migrating distributed system applications and multi-tier Internet services from local data center to Amazon EC2-like cloud, including the common migration problems observed and the key insights for solving the problems. In Section 5 we present the design of the CloudMig system, which provides both the configuration validation and installation automation to simplify the migration process.

2 Why Migration to Cloud is Complicated and Error-prone

There are some causes that make the migration process to Cloud complicated and error-prone. *First*, the computing environmental changes render many environment dependent configurations invalid. For example, as the database server is migrated from local data center to the Cloud, the IP address is possibly changed and this inevitably imposes the requirement of updating the IP address in all the components that depend on this database server. The migration process incurs large number of configuration update operations and even a single negligence of a single update may render the whole system out of operation. *Second*, the deployment of today's enterprise system consists of large number of different components. For example, for load balancing purpose, there may be multiple web servers and application servers in the systems. Thus the dependencies among the many components are rather complicated and can be broken very easily in the migration process. Sorting the dependency out to restore the normal operational status of the applications may take much more time than the migration process itself. *Third*, there are massive hidden controlling settings which may be broken inadvertently in the migration process. For example, the access controls of different components may be rumpled, which confront the system to the security threats. *Lastly*, the human operators in the complicated migration process may make many careless errors which are very difficult to identify. Overall, the complicated deployments, the massive dependencies, and the lack of automation make the migration process difficult and error-prone.

3 Related Work

Most of the existing migration approaches are either done manually or limited to only certain types of applications. For example, the suggestions recommended by Opencrowd are rather high level and abstract and lack the concrete assistances to the migration problem [4]. The solution provided by OfficetoCloud is only limited to the type of Microsoft Office products and does not even scratch the surface of large application migration [5]. We argue that a systematic and realistic study on the complexity of migrating large scale applications to Cloud is essential to direct the Cloud migration efforts. Furthermore, an automatic and operational approach is highly demanded to simplify and facilitate the migration process.

Nagaraja et al. [8] proposed a testbed for inserting faults to study the error behaviors. In our study, we study operator errors by migrating real practical applications from local data center to EC2. This forms a solid problem analysis context which motivates the effective solution for the migration problem. Vieira and Madeira [9] proposed to assess recoverability of database management systems through fault emulation and recovery procedure emulation. However, they assumed that human operators had the fault identification capability. In our work, we assume that human operators only have certain error identification capability but still cannot avoid errors.

Thus an automated configuration management system is highly demanded. There is already intensive research work on design and evaluation of interactive systems with human operators involved in the field of human computer interaction. For example, Maxion and Reeder in [7] studied the genesis of human operator errors and how to reduce them through user interface.

4 Migration Operations and Error Model

In this section, we describe a series of application migration practices conducted in migrating typical applications from a local data center to EC2 Cloud platform. We first introduce the experimental setup and then discuss the migration practices on representative applications in details and in particular, we focus on the most common errors made during the migration process. Based on our observations, we build the migration error model through the categorization of the migration errors.

4.1 Experiment Setup

Our experimental testbed involves both a local data center and EC2 Cloud. The local data center in College of Computing, Georgia Institute of Technology, is called "loki", which is a 12-node, 24-core Dell PowerEdge 1850 cluster. Because the majority of today's enterprise infrastructures are not virtualized, the physical to virtual (P2V) migration paradigm is the mainstream for migrating applications to virtualized cloud datacenters. In this work, we focus on P2V migration.

We deliberately selected representative applications as migration subjects. These applications are first deployed in the local data center and then operators are instructed to migrate from local data center to the Cloud. Our hypothesis is that application migration to the cloud is a complicated and error-prone process and thus a semi-automated migration validation system can significantly improve the efficiency and effectiveness of application migration. With the experimental setup across the local data center and Amazon EC2 platform, we are able to deploy moderate enterprise scale of applications for migration from a real local data center to the real Cloud platform and test the hypothesis under the setting of real workload, real massive systems, and real powerful Cloud.

We selected two types of applications in the migration case studies: Hadoop and RUBiS. These represent typical types of applications used in many enterprise computing systems today. The selection was made mainly by taking into account the service type, the architecture design and the migration content.

- Hadoop [1], as a powerful distributed computing paradigm, has been increasingly attractive to many enterprises to analyze large scale data generated daily, such as Facebook, Yahoo, etc. Many enterprises utilize Hadoop as a key component to achieve data intelligence. Because of its distributed nature, the more nodes participating in the computation, the more computation power is obtained in running Hadoop. Thus, when the computation resources are limited at local site, enterprises tend to migrate their data intelligence applications to Cloud to scale out the computation. From the aspect of service functionality, Hadoop is a very typical representation of data-intensive computation applications and thus the migration study on Hadoop provides us good referential value on data intensive application migration behaviors.

  Hadoop consists of two subsystems, map-reduce computation subsystem and Hadoop Distributed File System (HDFS), and thus migrating Hadoop from local data center to the Cloud includes both computation migration and file system migration or data migration. Thus it is a good example of composite migration. From the angle of architecture design, Hadoop adopts the typical master-slave structure in its two layers of subsystems. Namely, in map-reduce layer, a job tracker manages multiple task trackers and in the HDFS layer, a NameNode manages multiple DataNodes. Thus the dependency relationships among multiple system components form a typical tree structure. The migration study on Hadoop reveals the major difficulties or pitfalls in migrating applications with tree-style dependency relationships.

  In our P2V experiment setup, we deploy a 4-node physical Hadoop cluster, and designate one physical node to work as NameNode in HDFS or job tracker in map-reduce and four physical nodes as DataNode in HDFS or task tracker in map-reduce (the NameNode or job tracker also hosts a DataNode or task tracker). The Hadoop version we are using is Hadoop-0.20.2. The migration job is to migrate source Hadoop cluster to the EC2 platform into a virtual cluster with 4 virtual nodes.

- RUBiS [2] is an emulation of multi-tiered Internet services. We selected RUBiS as a representative case of large scale enterprise services. To achieve the scalability, enterprises often adopt the muti-tiered service architecture. Multiple servers are used for receiving Web requests, managing business logic, and storing and managing data: Web tier, application tier, and database tier. Depending on the workload, one can add or reduce the computation capability at a certain tier by adding more servers or removing some existing servers. Concretely, a typical three tier setup consists of using an Apache HTTP server, Tomcat application server and MYSQL database as the Web tier, application tier and database tier respectively.

  We selected RUBiS benchmark in our second migration case study by considering the following factors. First, Internet service is a very basic and prevalent application type in daily life. E-commerce enterprises such as EBay, usually adopts multi-tiered architecture as emulated by RUBiS to deploy their services and this renders RUBiS a representative case of Internet service architecture migration. Second, the dependency relationship among the tiers of multi-tiered services follows an acyclic graph structure, rather than a rigid tree structure, making it a good alternative in studying the dependency relationship preservation during the migration process. Third, the migration content of this type of application involves reallocation of application, logic and data and thus its migration provides a good case study on rich content migration. In the P2V experiment setup, one machine installs the Apache HTTPD server as the first

tier, and two machines install the Tomcat application server as the second tier, and two machines install the MYSQL database as the third tier.

In the following subsections, we introduce two migration case studies we have conducted: Hadoop migration and RUBiS migration, focusing mainly on configuration errors and installations errors. The configuration errors are our primary focus because they are the most frequent operator errors, some of which are also difficult to identify and correct. Installation errors can be corrected or eliminated by more organized installation steps or semi-automated installation tools with more detailed installation scripts and instructions.

We first discuss the typical configuration errors in each migration case study based on our error categorization model, which classifies the configuration errors into seven categories: dependency preservation error, network connectivity error, platform difference error, reliability error, shutdown and restart error, software and hardware compatibility error, and access control and security error. Then we describe the common installation errors across both case studies. We illustrate each category of errors by examples through selecting a subset of the typical errors observed in our experiments. Finally we present the statistical results on the error distributions in each case study and across case studies. This experimental analytic study of major errors lays a solid foundation for the design of a semi-automated migration validation system that offers effective configuration management.

4.2 Hadoop Migration Study

In the Hadoop migration case study, we migrate the source Hadoop application from the local data center to EC2 platform. This section discusses the typical configuration errors observed in this process.

Dependency Preservation. This is the most common error present in our experiments. Such a pitfall is very easy to make and very difficult to discover and may lead to disastrous results. According to the degree of severe impacts of this type of error on the deployment and migration, it can be further classified into four levels of errors.

The first level of errors is the "dependency preservation" error generated when the migration administrator fails to meet the necessity of dependency preservation checking. Even if the dependency information presents explicitly, lacking of enforcement to review the component dependency may lead to stale dependency information. For example, in our experiments, if the migration operator forgets to update the dependency information among the nodes in the Hadoop application, then the DataNodes (or task tracker) after migration will still initiate the connection with the old NameNode (or job tracker). This directly renders the system unoperational.

The second level of errors in Hadoop migration is due to incorrect formatting and typos in the dependency files. For example, a typo hidden in the host name or IP address renders some DataNodes to be unable to locate the NameNodes.

The third level of the dependency preservation error type is due to incomplete updates of dependency constraints. For example, one operator only updated the configuration files named "masters" and "slaves" which record the NameNode and list of DataNodes respectively. However, Hadoop dependency information is also located in some other configuration files such as "fs.default.name" in "core-site.xml" and "mapred.job.tracker" in mapred-site.xml. Thus Hadoop was still not able to boot with the new NameNode. This is a typical pitfall in migration, and is also difficult to detect by the operator because the operator may think that the whole dependency is updated and may spend intense efforts in locating faults in other locales.

The fourth level of the dependency preservation error type is due to inconsistency in updating the number of machines in the system. Often, an insufficient number of updated machines may lead to unexpected errors that are hard to debug by operators. For example, although the operator realizes the necessity to update the dependency constraints and also identifies all the locations of constraints on a single node, the operator may fail to update all the machines in the system, which are involved in the system-wide dependency constraints. For example, in Hadoop migration, if not all the DataNodes update their dependency constraints, the system cannot run with the participation of all the nodes.

Network Connectivity Bearing the distributed computing nature, Hadoop involves intensive communication across nodes in the sense that the NameNode keeps communication with DataNodes and job tracker communicates with task tracker continuously. Thus for such system to work correctly, inter-connectivities among nodes become an indispensible prerequisite condition. In our experiments, operators showed two types of network connectivity configuration errors after migrating Hadoop from the local data center to EC2 in the P2V migraton paradigm.

The first type of such error is that some operators did not set the network to enable all the machines to be able to reach each other over the network. For example, some operators forgot to update the file "/etc/hosts" and led to IP resolvement problems. The second type of such error is local DNS resolution error. For example, some operators did not set the local DNS resolution correctly, which led to the consequence that only the DataNodes residing in the same host as the master node were booted after the migration.

Platform Difference The platform difference between EC2 Cloud and local data center also creates some errors in migrating applications. These errors can be classified into three levels: *security*, *communication*, and *incorrect instance operation*. In our experiment, when the applications are hosted in the local data center, the machines are protected by the firewalls, and thus even if the operators set simple passwords, the security is complemented by the firewalls. However, when the applications are migrated into the public Cloud, the machine can experience all kinds of attacks and thus too simple passwords may render the virtual hosts susceptible to security threats. The second level of the platform difference error type is related to the communication setting difference between cloud and local data center. For example, such error may occur after the applications are migrated into EC2 Cloud, if the communication between two virtual instances is still set in the same way as if the applications were hosted in the local data center. Concretely, for the operator in one virtual instance to ssh another virtual instance, the identify file which is granted by Amazon must be provided. Without the identify file, the communication within virtual instance cannot be set correctly. The third level of the platform difference error type is rooted in the difference between virtual instance management infrastructures. In the experiments, there were operators who terminated an instance but his actual intention is to stop the instance. In EC2 platform, termination of an instance will lead to the elimination of the virtual instance from Cloud and thus all the applications installed and all the data stored within the virtual instance are lost if data is not backed up in persistent storage like Amazon Elastic Block storage. Thus, this poses critical risks on the instance operations, because a wrong instance operation may wipe out all the applications and data.

Reliability Error: In order to achieve fault tolerance and performance improvements, many enterprise applications like Hadoop and multi-tiered Internet services replicate its data or components. For example, in Hadoop, data is replicated in certain number of DataNodes, while in multi-tiered Internet services, there may exist multiple application servers or database servers. Thus after the migation, if the replication degree is not set correctly, either the migrated application fails to work correctly or the fault tolerance level is compromised. For example, in the experiments, there were cases in which the operator made errors that set the replication degree more than the total number of DataNodes in the system. The reliability errors are sometimes latent errors.

Shutdown and Restart: This type of error means that the shutdown or restart operation in the migration process may cause errors if not operating correctly. For example, a common data consistency error may occur if Hadoop is incorrectly shuts down the HDFS. More seriously,a shutdown or restart error sometimes may compromise the source system. In our experiment, when the dependency graph was not updated consistently and the source cluster was not shut down completely, the destination Hadoop cluster initiated to connect to the source cluster and acted as the client to connect to the source cluster. As a result, all the operations issued by the destination cluster actually manipulated the data in the source cluster and thus the source cluster data was contaminated. Such errors may create disastrous impacts on the source cluster and are dangerous if the configuration errors are not detected in time.

Software and Hardware Compatibility: This type of error is less common in Hadoop migration than in RUBiS migration partly because Hadoop is built on top of Java and thus has better interoperability and also Hadoop involves a relatively smaller number of different components than RUBiS. Sometimes, the difference in software versions may lead to errors. For instance, the initial Hadoop version selected by one operator was Hadoop 0.19, which showed bugs in the physical machine. After the operator turned to the latest 0.20.2 version, the issue disappeared.

Access Control and Security: It is noted that a single node Hadoop cluster can be set and migrated without root access. However, because a multi-node Hadoop cluster needs to change the network inter-connectivity and solve the local DNS resolution issue, the root access privilege is necessary. One operator assumed that the root privilege was not necessary for multi-node Hadoop installation and was blocked due to the network connectivity problem for about one hour and then sought help for access to the root privilege.

4.3 RUBiS Migration Study

In the RUBiS migration experiments, we migrate a RUBiS system with one web server and two application servers and two database servers from the local data center to EC2 Cloud. We below discuss the configuration errors present in the experiments in terms of the seven types of error categories.

Dependency Preservation: Similar to Hadoop migration, the dependency preservation error type is also the most common error in RUBiS migration. Because RUBiS has more intensive dependency among different components than Hadoop, operators made more configuration errors in the migration. For different tiers of a RUBiS system to run cooperatively, dependency constraints need to be specified explicitly in relevant configuration locales. For example, for each Tomcat server, its relevant information needs to be recorded in the configuration file named " workers.properties" in Apache HTTPD server. The MYSQL database server needs to be recorded in the RUBiS configuration file named "mysql.properties". Thus an error

in any of these dependency configuration files will lead to the operation error. In our experiments, operators made different kinds of dependency errors. For example, some operator migrated the application but forgot to update the Tomcat server name in workers.properties. As a consequence, although the Apache HTTPD server was running correctly, RUBiS was not operating correctly because the Tomcat server could not be connected. One operator could not find the configuration file location to update the MYSQL database server information in RUBiS residing in the same host as Tomcat and this led to errors and the operator therefore gave up the installation.

Network Connectivity: Relative to Hadoop migration, there is less node interoperability in a multi-tiered system like RUBiS, and different tiers present less needs on network connectivity, thus the network connectivity configuration errors are less frequently seen in RUBiS migration. One typical error was seen when the operator was connecting the Cloud virtual instance, he forgot to provide the identity file to enable two virtual instances to connect via ssh.

Platform Difference : This error type turns out to be a serious fundamental concern in RUBiS migration. Because sometimes the instance rebooting operation may change the domain name, public IP and internal IP, even if the multi-tiered service is migrated successfully, a rebooting operation may render the application to service interruption. One operator finished the migration and after fixed a few configuration errors, the application was working correctly in EC2. After we turned off the system on EC2 for one day and then rebooted the service, we found that because the domain name had totally changed, all of the IP addresses or host name information in configuration files needed to be updated.

Reliability Error: Due to the widely used replication in enterprise systems, it is typical that the system may have more than one application server and/or more than one database server. One operator spelt the name wrong for the second Tomcat server, but because there remained a working Tomcat server due to replication, the service was still going on without interruption. However, a hidden error as such was hidden inside the system and it may cause unexpected errors that could lead to detrimental damage and yet is hard to debug and correct. This further validates our argument that configuration error detection and correction tools are critical for cloud migration validation.

Shutdown and Restart: This type of error shows that incorrect server start or shutdown operation in multi-tiered services may render the whole service unavailable. For example, the Ubuntu virtual instance selected for the MYSQL tier has a different version of MYSQL database installed by default. One operator forgot to shut down and remove the default installation first before installing the new version of MYSQL and thus caused errors. The operator spent about half an hour to find the issues and fixed them. Also we observed a couple of incidents where the operator forgot to boot the Tomcat server first before the shutdown operation, thus causing errors that are time consuming to debug.

Software and Hardware Compatability: this type of error also happens frequently in RUBiS migration. The physical machine is 64 bits, while one operator selected the 32 bits version of mod_jk (the component used to forward the HTTP request from Apache HTTPD server to Tomcat server) and thus incompatibility issues occured. The operator was stuck for about two hours, and finally identified the version error. After the software version was changed into 64 bits, the operator successfully fixed the error. A similar error was observed where an operator selected an arbitrary MYSQL version which took about one hour for the failed installation and then switched to a newer version before finally successfully installed the MYSQL database server.

Access Control and Security: This type of error also occurs frequently in RUBiS migration. For example, the virtual instance in EC2 Cloud bears the default feature of all ports closed. To enable the SSH operation possible, the security group where the virtual instance resides must open the corresponding port 22. Also one operator configured the Apache HTTPD server successfully but the Web server was unable to connect through port 80 and it took about 30 mins to identify the restrictions from EC2 documentation. Similar errors also happened for port 8080 which was for accessing Tomcat server. Another interesting error is that one operator set up the Apache HTTPD server, but forgot to set the root directory to be accessible and thus the index.html was not accessible. The operator reinstalled the HTTPD server but still did not discover the error. With the help of our configuration assistant, this operator finally identified the error and changed the access permission and fixed the error. We also found that operators also made errors in granting privileges to different users and one case was solved by seeking help in the MYSQL documentation.

4.4 Installation Errors

In our experiments-based case studies, we observe that operators may make all kinds of errors in installation or redeployment of the applications in Cloud. More importantly, these errors seem to be common across all types of applications. In this section we classify these errors into the following categories: **Context information error**: This is a very common installation error type. A typical example is that operators forget the context information they have used in the past installation. For example, the operators remembered the wrong path to install their applications and have to reinstall the applications from

scratch. Also if there are no automatic installation scripts or an incorrect or incomplete installation script is used, it can be a very frustrating experience with the same procedures repeated again and again. If the scale of the computing system is large, then the repeated installation process turns out to be a heavy burden for system operators. Thus a template based installation approach is highly recommended.

Environment compatibility error : In this migration case study, before any application can be installed, the computing environment compatibility needs to be ensured at both the hardware and software level. For example, there were migration failures created due to the small available disk space in virtual instance in migrating RUBiS. A similar errors is that the operator created a virtual instance with 32 bits operating system, while the application was a 64 bits version. Thus, it is necessary to check the environment compatibility before the application installation starts. An automatic environment checking process helps to reduce the errors caused by incorrect environment settings.

Prerequisite resource checking error : This type of error is originated from the fact that every application depends on a certain set of prerequisite facilities. For example, the installations of Hadoop and Tomcat server presume the installation of Java. In the experiments, we observed that the migration or installation process were prone to be interrupted by the ignorance of installing prerequisite standard facilities. For example, the compilation process needs to restart again due the lack of "gcc" installation in the system. Thus, a complete check-list of the prerequisite resources or facilities can help us reduce the interruptions of the migration process.

Application installation error: this error is the most common error type experienced by the operators. The concrete application installation process usually consists of multiple procedures. We found that the operator made many repeated errors even when the installation process for the same application was almost the same. For example, operators forgot the building location of the applications. Thus a template based application installation process will help facilitate the installation process.

4.5 Migration Error Distribution Analysis

In this section, we analyze the error distributions for each specific application and the error distribution across the applications.

Figure 1 and Figure 2 show the number of errors and percentage of error distribution in the Hadoop migration case study. In both figures, the X-axis indicates the error types as we analyzed in the previous sections. The Y-axis in Figure 1 shows the number of errors for each particular error type. The Y-axis in Figure 2 shows the share of each error type in terms of the percentage over the total number of errors. In this set of experiments, there were a total of 24 errors and some errors cause violation in multiple error categories. In comparison, the dependency preservation error happened most frequently. 42% of the errors belong to this error type with 10 occurrences. Operators typically made all four levels of dependency preservation errors as we discussed in Section 4.2. These kinds of errors took a long time for operators to detect. For example, an incomplete dependency constraint checking error took one operator two and a half hours to identify the cause of the error and fix it. Network connectivity error

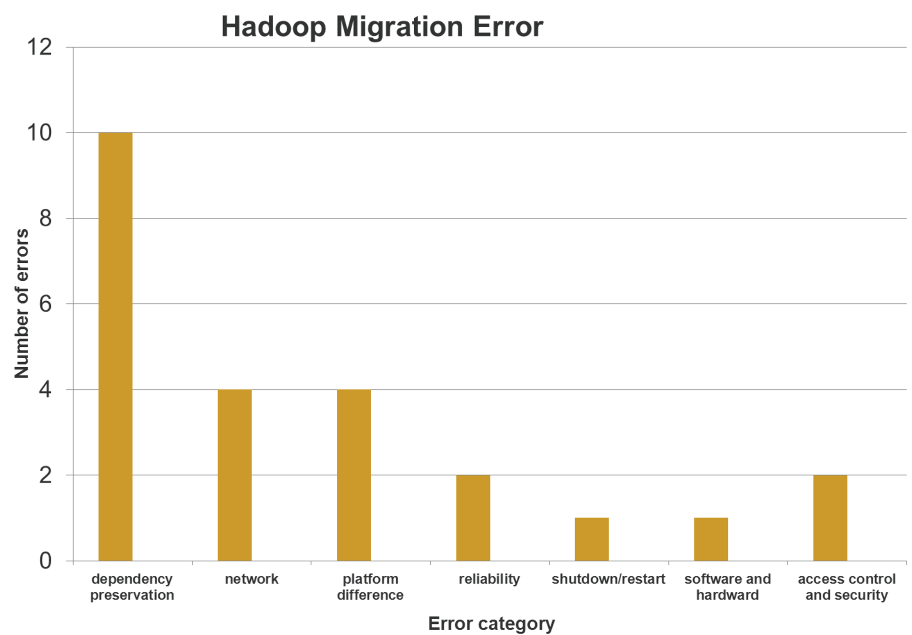
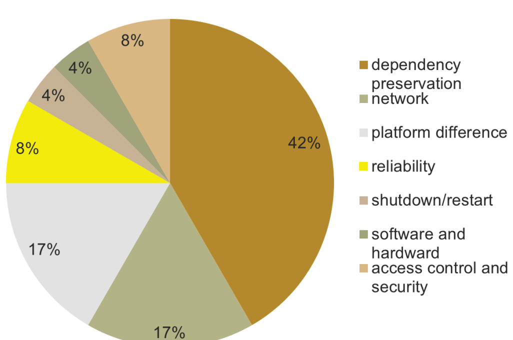


Figure 1. Hadoop migration error



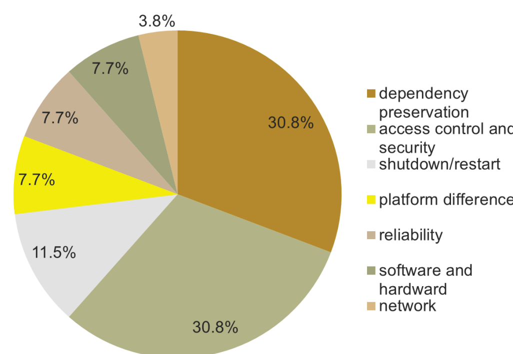


Figure 2. Hadoop migration error distribution. The legend lists the error types in the decreasing frequency order.

Figure 3. RUBiS error distribution. The legend lists the error types in the decreasing frequency order.

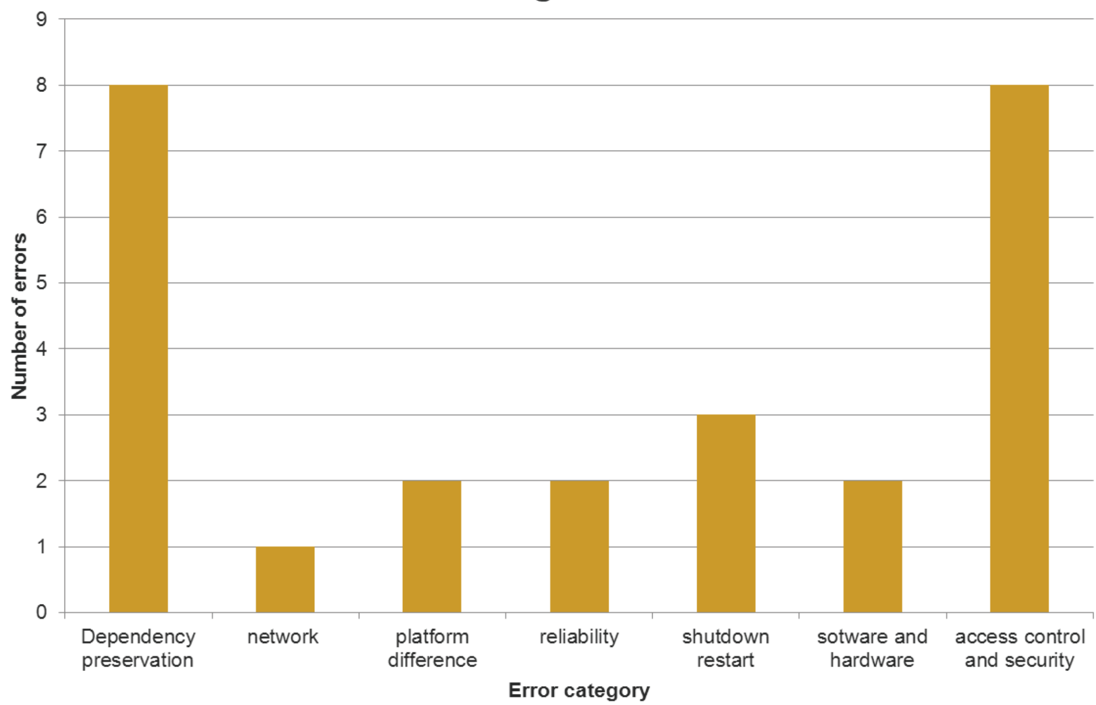


Figure 4. RUBiS migration error

and platform difference error were the next most frequent error types, each taking 17% of the total errors. Network connectivity errors included local DNS resolution and IP address update errors. One typical platform difference error was that the termination of an instance led to the data loss. Interesting to note is that these three types of errors take 76% of the total errors and are the dominating types of the error occurrences observed in the experiments we conducted.

Figure 4 and Figure 3 show the number of error occurrences and the percentage of error distribution for RUBiS migration case study respectively. There were a total of 26 error occurrences observed in this process and some errors fall into several

error categories. The dependency preservation error and access control and security errors were the two most frequent error types, each with 8 occurrences, taking 31% of the total erorrs. Together, both error types covered 62% of all the errors and dominated the error occurrences. It is interesting to note that the distribution of errors in the RUBiS migration case study was very different from the distribution in the Hadoop migration case study. For example, the number of access and security errors in RUBiS was 4 times the number of errors of this type in Hadoop migration. This is because RUBiS migration demanded the correct access control settings for many more entities than Hadoop. Not surprisingly, the majority of the access control errors were file access permission errors. This is because changing the file access permission is a common operation in setting up web services and sometimes operators forgot to validate whether the access permissions were set correctly or not. Also when there were errors and the system could not run correctly, the operators often ignored the possibility of this type of simple errors and thus led to longer time spent on error identification. For example, one error of this type took more than 1 hour to identify. Also there were more ports to open in RUBiS migration than in Hadoop migration, which also led to the high frequency of access control errors in RUBiS migration. RUBiS migration presented more software and hardware compatibility errors than Hadoop migration because the number of different components that were involved in RUBiS application is, relatively speaking, much more than in the typical Hadoop migration. Similarly, there were more "shutdown/restart" errors in the RUBiS migration. On the other hand, Hadoop migration presented more network connectivity errors and platform difference errors than RUBiS migration, because Hadoop nodes require more tightly coupled connectivity than the nodes in RUBiS. For example, the master node needs to have direct access without password control to all of its slave nodes.

Figure 5 and Figure 6 summarize across Hadoop migration and RUBiS migration case studies by showing the number of error occurrences and the percentage of error distribution, respectively.The dependency preservation errors are the most frequent error occurrences and accounted for 36% of the total errors. In practice, this was also the type of error that on average took the longest time to identify and fix. This is primarily because dependency constraints are widely distributed among system configurations, it is very prone to be broken by changes to the common configuration parameters. The second biggest error source was the "access control and security" errors, which accounted for 20% of the total number of error occurences. It was very easy for operators to change the file permissions to incorrect settings or some other habits which were fitting in local data center might render the application susceptible to security threats in the Cloud environment. The operational or environmental differences between Cloud and local data centers formed the third largest source of error, accounting for 12% of all the errors. Many common operations in local data center might lead to errors in Cloud if no adjustments to Cloud environment were made. These three types of errors dominated the error distribution, and accumulatively accounted for 68% of the total errors. In addition to these three types of errors, network connectivity was also an important source of errors, accounting for 10% of the total errors, because of the heavy inter-nodes operations in many enterprise applications today. The rest of errors accounted for 32% of the total errors. These error distributions provide a good reference model for us to build a solid testbed to test the design of our CloudMig migration validation approach to be presented in the subsequent sections of this paper. We argue that a cloud migration validation system should be equipped with an effective configuration management component that not only provides a mechanism to reduce the configuration errors, but also equips the system with active configuration error detection and debugging as well as semi-automated error correction

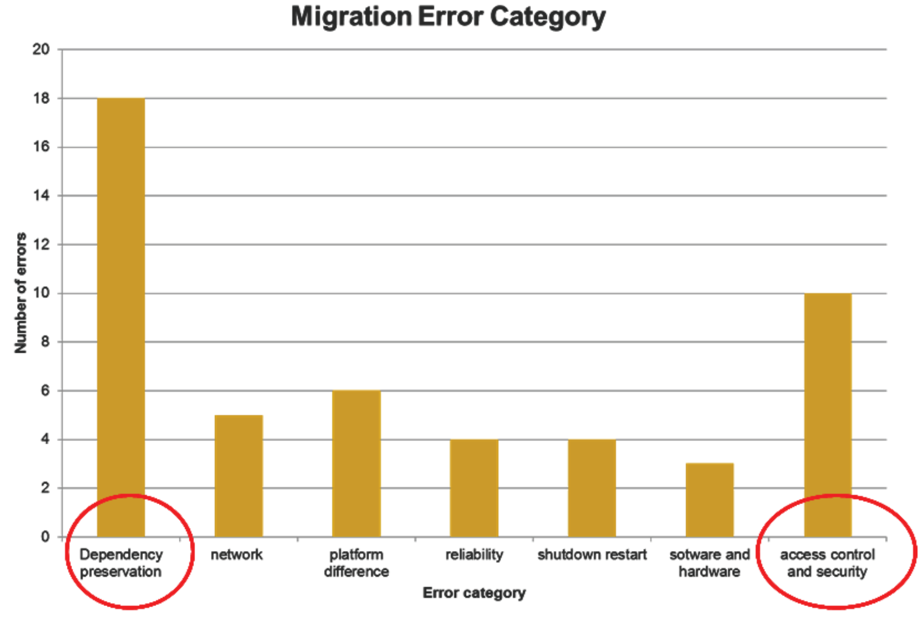


Figure 5. Overall migration error

and repairs.

5 Migration Validation with CloudMig

The case studies showed that the installation mistakes and configuration errors were the two major sources of errors in migrating applications from local data centers to Cloud. Thus a migration management framework is highly recommended to provide the installation automation and configuration validation. We present the design of CloudMig, a semi-automated configuration management system, which utilizes a "template" to simplify the large scale enterprise system installation process and utilizes a "policy" as an effective means to capture configuration dependency constraints, validate the configuration correctness, and monitor and respond to the configuration changes.

Figure 6. Overall migration error distribution. The legend lists the error types in the decreasing frequency order.

The architecture design of the CloudMig system aims at coordinating the migration process across different data centers by utilizing template-based installation procedures to simplify the migration process and utilizing policy-based configuration management to capture and enforce configuration related dependency constraints and improve migration assurance.

The first prototype of CloudMig configuration management and validation system consists of four main components: the centralized configuration management engine, the client-based local configuration management engine, the configuration template management tool and the configuration policy management tool. The template model and the configuration policy model form the core of CloudMig for semi-automated installation and configuration validation system. In the subsequent sections we will briefly describe the functionality of each of these four components.

5.1 Configuration Template Model

CloudMig uses a template as an effective mechanism to simplify the installation process. Template is a pre-formatted script-based example file containing place holders for dynamic and application-specific information to be substituted at application migration time for concrete use.

In CloudMig, the installation and configuration management is operating in the unit of the application. That is, each application corresponds to a template set and a validation policy set. The central management server is responsible to manage the collection of templates and configurations on a per application basis and provides migration planning for the migration process.

Recall that in the observations obtained from our migration experiments in Section 4, one big obstacle and source of errors in application migration is the installation and configuration process which is also a recurring process in system deployment and application migration. We propose to use the template approach to reduce the complexities of the installation process and reduce the chances of errors. An installation template is defined by an installation script with place holders for dynamic and application specific information. Templates simplify the recurring installation practice of particular applications by

substituting the dynamic information with new values. For example, in an enterprise system with 100 nodes, there will be multiple applications ranging from MYSQL database nodes, Tomcat application server nodes, to Hadoop distributed system nodes and so forth. Distributed applications may span and extend to more nodes on demand to scale out. For each application, its installation templates are stored in the installation template repository. These templates are sorted by the application type and an application identifier. The intuitive idea of template is that through information abstraction, the template can be used and refined for many similar nodes through parameter substitution to simplify the installation process for large scale systems. For example, if a Hadoop system consists of 100 DataNodes, then only a single installation template is stored in the installation template repository and each DataNode will receive the same copy of the installation template with only parameter substitution efforts needed before running the installation scripts to set up the DataNode component in each individual node. The configuration dependency constraints are defined in the policy repository to be described in the next subsection. CloudMig classifies the templates into the following four types:

1. Context dictionary: This is the template specifying the context information about the application installation. For example, the installation path, the preassumed Java package version, etc. A context dictionary template can be as simple as a collection of the key-value pairs. Users specify the concrete values before a particular application installation. Dynamic place holders for certain key context information achieve the installation flexibility and increase the ability to find out the relevant installation information in the presence of system failures.

2. Standard facility checklist template: This is the script template to check the prerequisites to install the application. Usually these are some standard facilities, such as Java or OpenSSH. Typical checklists include those for verifying the Java path setting, checking installation package existence, and so on. These checklists are common to many applications and are prerequisites for the success of installing the applications and thus performing a template check before the actual installation can effectively reduce the errors caused by ignorance of the checklist items. For example, both Hadoop and Tomcat server rely on the correct Java path setting and thus the correct setting of Java path is the prerequisite of successfully installing these two applications. In CloudMig, we collect and maintain such templates in a template library, which is shared by multiple applications. Running the checklist validation check can effectively speed up the installation process by reducing the amount of errors caused by carelessness on prerequisites.

3. Local resource checklist template: This is the script template to check the hidden conditions for an application to be installed. A typical example is to perform the check of whether or not there is enough available disk space quota for a given application. Similarly, such resource checklist templates are also organized by application type and application identifier in the template library and utilized by the configuration management client to reduce the local installation errors and installation delay.

4. Application installation template: This is the script template used to install a particular application. The context dictionary is included as a component of the template. Organizing installation templates simplifies the installation process and thus reduces the overhead in recurring installations and migration deployments.

5.2 Configuration Policy Model

In this section, we first introduce the basic concept of configuration policy, which plays the key role in capturing and specifying configuration dependency constraints and monitoring and detecting configuration anomalies in large enterprise application migration. Then we introduce the concept of continual query (CQ) and the design of a CQ enabled configuration policy enforcement model.

5.2.1 Modeling Dependency Constraints with Configuration Policies

A configuration policy defines an application-specific configuration dependency constraint. Here is an example of such constraints for RUBiS: for each Tomcat server, its relevant information needs to be specified explicitly in the configuration file named “workers.properties” in Apache HTTPD server. Configuration files are usually application-specific and usually specify the settings of the system parameters, the dependencies among the system components and thus directly impact the way of how the system is running. As enterprise applications scale out, the number of components may increase rapidly and the correlations among the system components evolve with added complexity. In term of complexity, configuration files for a large system may cover many aspects of the system configuration, ranging from host system information, to network setting, to security protocol and so on. Any typo or error may disable the operational behavior of the whole application system

as we showed and analyzed in the previous experiments. Configuration setting and management are usually a long term practice, starting from the time when the application is set up until the time when the application is ceased its use. During this long application life cycle, different operators may be involved in the configuration management practices and operate on the configuration settings based on their understandings, thus it further increases the probability of errors in configuration management. In order to fully utilize resources, enterprises may bundle multiple applications to run on top of a single physical node, and the addition of new applications may necessitate the need to change the configurations of existing applications. Security threats such as viruses, also pose demands to effective configuration monitoring and management.

In CloudMig, we propose to use policy as an effective means of ensuring the constraints of configurations to be captured correctly and enforced consistently. A policy can be viewed as a specialized tool to specify the constraints on the configuration of a specific application. It specifies the constraints to which the application configuration must conform in order to assure that the whole application is migrated correctly to run in the new environment. For example, in the Hadoop system, the configuration constraint that "the replication degree cannot exceed the number of DataNodes" can be represented as a Hadoop specific configuration policy. The basic idea of introducing the policy-based configuration management model is that if operators are equipped with a migration configuration tool to define the constraints that configuration must follow in the form of policies, then running the policy enforcement checks at a certain frequency will help to detect and eliminate certain types of errors, even although errors are unavoidable. Here are a few configuration policy examples that operators may have in migrating a Hadoop system.

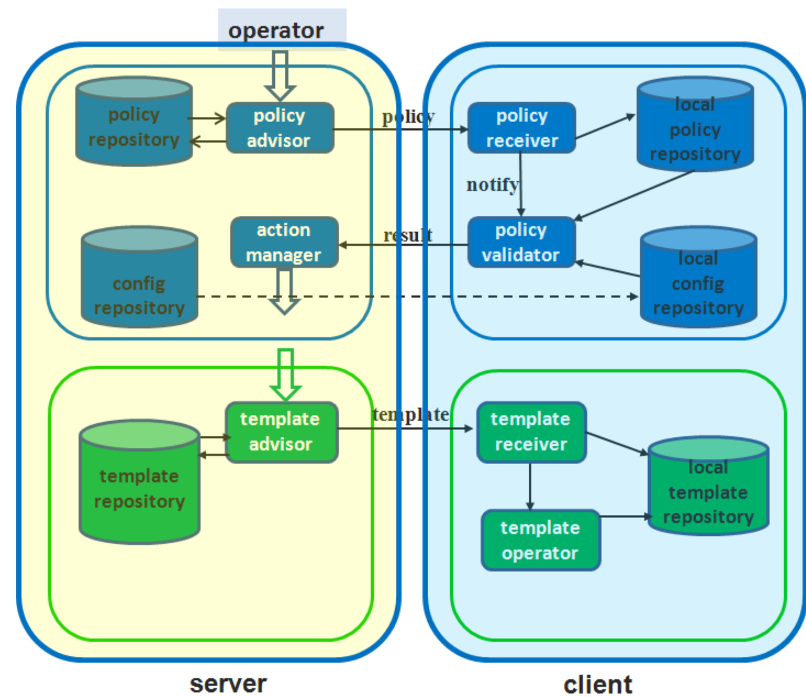


Figure 7. CloudMig Architecture Design

1. The replication degree can not be larger than the number of DataNodes
2. There is only one master node
3. The master node of Hadoop cluster should be named "dummy1"
4. The task tracker node should be named "dummy2"

As the system evolves and the configuration repository grows, performing such checking manually will become a heavy and error-prone process. For example, in enterprise Internet service systems, there may be hundreds of nodes, and the configuration of each node needs to follow certain constraints. For load balancing purpose, different Apache HTTPD servers correspond to different sets of Tomcat servers. Incorrect setting of relevant configuration entries will directly lead to an unbalanced system and even cause the system to crash when workload burst happens. With thousands of configuration entries, hundreds of nodes, and many applications, it is impractical if not impossible to perform manual configuration correctness checking and error correction. We argue that a semi-automated configuration constraint checking framework can greatly simplify the migration configuration and validation management of large scale enterprise systems. In CloudMig, we advocate the use of continual query as the basic mechanism for automating the configuration validation process of operator-defined configuration policies. In the next section we will describe how CQ-enabled configuration policy management engine can improve the error detection and debugging efficiency, thus reducing the complexity of migrating applications from a local data center to Cloud.

5.2.2 Continual Query Based Policy Model

In CloudMig, we propose a continual query based policy specification and enforcement model. A continual query (CQ) [6] is defined as a triple in the form of (Query, Trigger, Stop). A continual query (CQ) can be seen as a standing query, in which the trigger component specifies the monitoring condition and is being evaluated periodically upon the installation of the CQ and whenever the trigger condition is true, the query component will be executed. The Stop component defines the condition to terminate the execution of the CQ. Trigger condition can be either time-based or content-based, such as "checking the free disk space every hour or trigger a configuration action when the free disk space is less than 1GB".

In CloudMig, we define a policy in the format of continual query and refer to the configuration policy as the Contiual Query Policy (CQP), denoted by : CQP(policyID, appName, query, trigger, action, stopCondition). Each element of the CQP is defined as follows:

1. *policyID* is the unique numeric identifier of the policy.
2. *appName* is the name of the application that is installed or migrated to the host machine.
3. *query* refers to the search of matching policies and the execution of policy checking. The query can be a Boolean expression over a simple key-value repository or SQL-like query or XQuery on a relational database of policies.
4. *trigger* is the condition upon which the policy query will be executed. Triggers can be classified into time-based or content-based.
5. *action* indicates the action to be taken upon the query results. It can be a warning flag in the configuration table or an warning message sent by email or displayed on the command line of an operator's terminal.
6. *stopCondition* is the condition upon which the CQP will stop to execute.

An example CQ-based policy is to check whether the replication degree is larger than the number of DataNodes in Hadoop prior to migration or changing the replica factor (replication degree) or reducing the number of DataNodes. Whenever the check returns a true value, send an alert to re-configure the system. Clearly, the query component is responsible for checking if the replication degree is larger than the number of DataNodes in Hadoop. The trigger condition is Hadoop migration or changing the replica factor (replication degree) or reducing the number of DataNodes. The action is defined as re-configuration of the Hadoop system upon the true value of the policy checking. In CloudMig, we introduce default stop condition of one month for all CQ-enabled configuration policies.

5.3 CloudMig Server side Template Management and Policy Management

CloudMig aims at managing the installation templates and configuration policies to simplify the migration for large scale enterprise systems which may be comprised of thousands of nodes with multi-tier applications. Each application has its own configuration policy set and installation template set and the whole system needs to manage a large collection of configuration policies and installation templates. The CloudMig server side configuration management system helps to manage the large collection of templates and configuration policies effectively by providing system administrators (operators) with convenient tools to operate on the templates and policies. Typical operations include policy or template search, indexing, application specific packaging and shipping, to name a few. Detaching the template and policy management from individual application and utilizing a centralized server also improves the reliability of CloudMig in the presence of individual node failures.

In CloudMig, the configuration management server operates at the unit of a single application. Each application corresponds to an installation template set and a configuration validation policy set. The central management server is responsible for managing the large collection of configurations on a per application basis and providing migration planning to speed up the migration process and increase the assurance of application migration. Concretely, the configuration management server mainly coordinate the tasks of the installation template management engine and the configuration policy management engine.
Installation Template Management Engine.

As shown in Figure 7, the installation template management engine is the system component which is responsible for creating template, update template, advise the template for installation. It consists of a central template repository and a template advisor. The template repository stores and maintains the template collections of all the applications. The template advisor provides the operators with the template manipulation capabilities such as creating, updating, deleting, searching and

Table 1. Migration Error Detection Rate

Migration Type	Error Detection Rate
Hadoop migration	83%
RUBiS migration	55%
all migrations	70%

indexing templates over the template repository. On a per application basis, operators may create an application template set, add new templates to the set, update templates from this set or delete templates. The template advisor assumes the job to search and dispatch templates for new installations and propagate template updates to corresponding application hosting nodes. For example, during the process of RUBiS installation, for a specific node, the template advisor dispatches the appropriate template depending on the server type (web server, application server or database server) and transmits (ships) the new installation set to the particular node.

Concretely, for each application, the central installation template management engine builds the context library which stores all the source information in the key-value pairs, and selects a collection of standard facility checklist templates which apply to the particular application, and pick a set of local resource checklist templates as the checklist collection for the application, and finally builds the specific application installation template. The central management engine then bundles the collections of templates and policies for the particular application and transmits the bundle to the client installation template manager to start the installation instance.

Configuration Policy Management Engine.
As the central management unit for the policies, the policy engine consists of four components: policy repository, configuration repository, policy advisor, and action manager. Together they cooperate to provide the service to create, maintain, dispatch and monitor policies and execute the corresponding actions based on the policy execution results. Concretely, we below describe the different components of the policy engine:

1. The policy repository is the central store where all the policies for all the applications are maintained. It is also organized on a per application basis. Each application corresponds to a specific policy set. This policy set is open to addition, update, or delete operations. Each policy corresponds to a constraint set on the application.

2. The policy advisor works on the policies in the policy repository directly and provides the functionalities for application operators to express the constraints in the form of CQ-based policy. Application operators creates policies through this interface.

3. The configuration repository stores all the configuration files on a per application basis. It ships the configurations from the CloudMig configuration server to the local configuration repository on the individual node (cient) of the system.

4. The action manager handles the validation results from the policy validator running on client and triggers the corresponding action based on certain policy query result, in the form of an alert through message posting or email or other notification methods.

5.4 CloudMig Configuration Management Client

The CloudMig configuration management client is running at each node of a distributed or multi-tier system, which is responsible for managing the configuration policies related to the node locally. Corresponding to the CloudMig configuration management engine at the server side, CloudMig client works as a thin local manager for the templates and policies which only apply to a particular node. A client engine mainly consists of two components: client template manager and client policy manager.

Client Template Manager.
Client template manager manages the templates for all the applications installed in the host node on per application basis. It consists of three components: template receiver, template operator and local template repository. The template receiver receives the templates from the remote CloudMig configuration management server and delivers the templates to local template manager. The local template manager installs the application based on the template with necessary substitution operations. The local template manager is also responsible for managing the local template repository which stores all the templates for the applications that reside at this node.

The concrete process of template based installation works as follows: after the client template manager receives the collection of installation templates from the server side installation template management engine, it will run the local resource checklist templates first to detect if there are any prerequisite checklist items which are not met. For example, it checks if the available disk space is less than the amount needed to install the application, or if the user has the access permissions to the installation path, etc. Next, the standard facility checklist template will run to detect if all the standard facilities are installed or not. Finally, the dynamic information in application specific templates are substituted and the context dictionary is integrated to run this normal installation process.

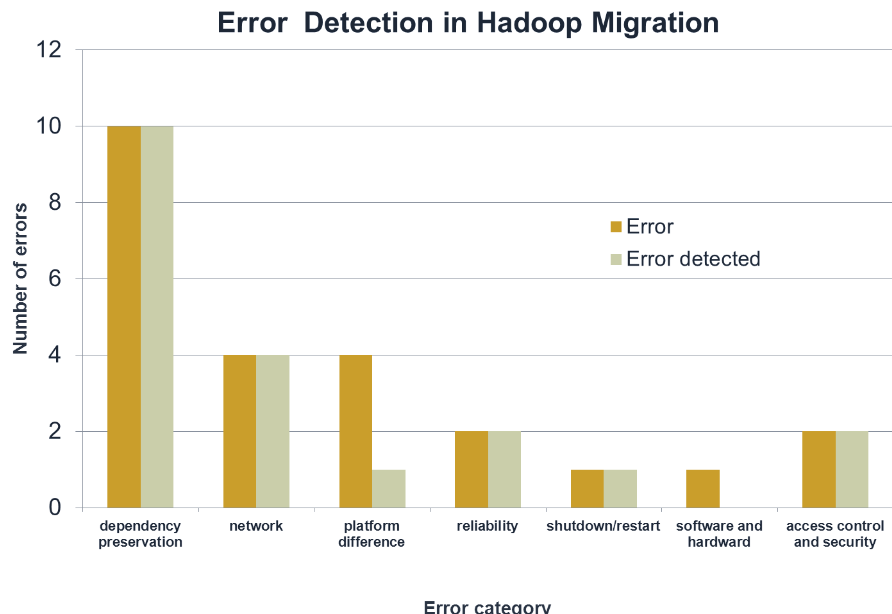


Figure 8. Hadoop error detection

Client Policy Manager.

There is a client policy manager residing together with the host node to manage the policies for the local node. It mainly consists of policy receiver, policy validator, local policy repository and local config repository. The policy receiver receives the policies transmitted from the policy advisor in the central policy server, and stores the policies in the local policy repository. The local config repository receives the configuration data directly from the central config repository. The local policy validator runs each policy. It retrieves the policy from local policy repository and searches the related configuration data to run the policy upon the configuration data. The policy validator transmits the validation results to the action manager in

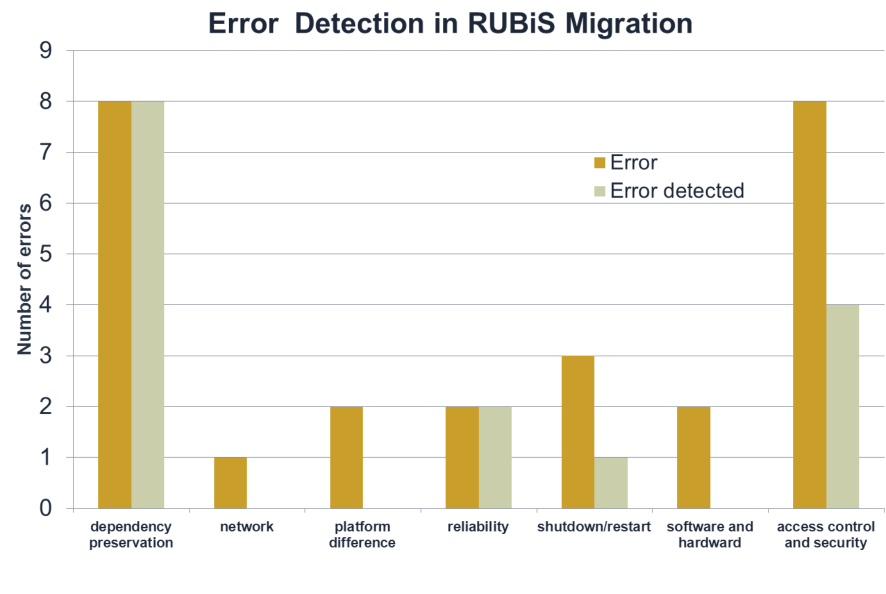


Figure 9. RUBiS error detection

the central server to take the alert actions.

6 Case Studies with CloudMig

We run CloudMig with the same set of operators on the same set of case studies after the manual migration process is done (recall Section 4). We count the number of errors that are detected by CloudMig configuration management and installation automation system. We show through a set of experimental results below that CloudMig overall achieves high error detection rate.

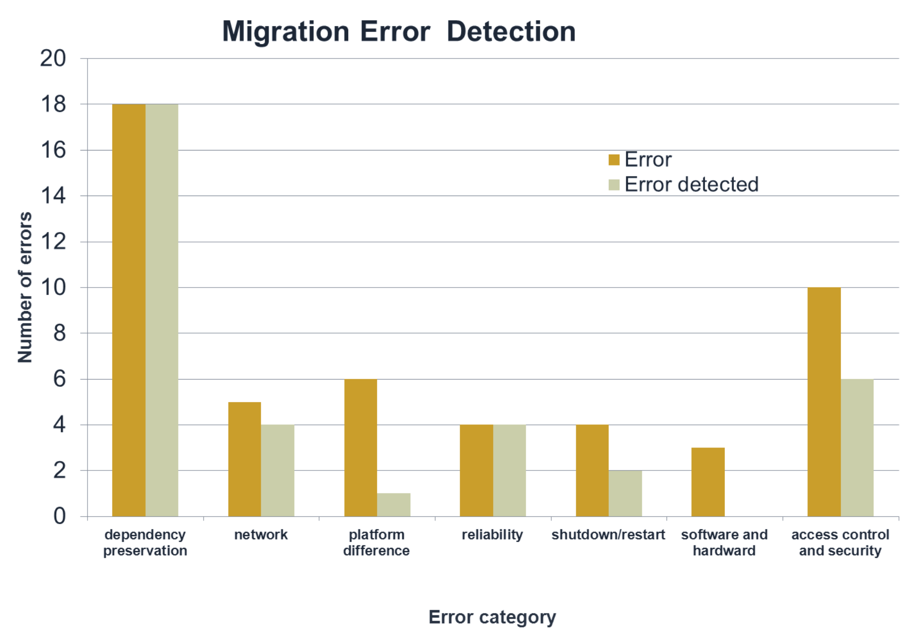


Figure 10. Overall migration error detection

Figure 8 shows the error detection results for Hadoop migration case study. As one can see, the configuration checking system can detect all the dependency preservation errors, network connectivity errors, shutdown restart errors, and all the access control errors. This confirms the effectiveness of the proposed system, in that it can detect the majority of the configuration errors. The two types of error that can not be fully detected are platform difference error and software/hardware compatibility errors. For platform difference errors, this is because the special property of the platform difference error requires the operators to fully understand the uniqueness of the particular Cloud platform first. As long as the operator understands the platform sufficiently, for example, by lessons learned from others or policies shared by others, we believe that such errors can be reduced significantly as well. The reason that current implementation of CloudMig cannot detect software/hardware compatibility errors notably is due to the quality of the default configuration data which lacks of application-specific software/hardware compatibility information. Although in the first phase of implementation, we mainly focus on the configuration checking triggered by the original configuration data, we believe that as operators weave more compatibility policies into CloudMig policy engine, such type of errors can also be reduced significantly. As Table 1 shows, totally CloudMig could detect 83% of the errors in Hadoop migration.

Figure 9 shows the error detection result for RUBiS migration case study. In this study, we can see that CloudMig can detect all the dependency preservation errors and reliability errors.

However, because multi-tiered Internet service system involves a higher number of different applications, it leads to more complicated software/hardware compatibility issues compared to the case of Hadoop migration. In the experiments reported in this paper we are focusing on the configuration driven by the default configuration policies, which lacks of adequate software/hardware compatibility policies for RUBiS, thus CloudMig system did not detect the software/hardware errors. On the other hand, this result also indicates that in the RUBiS migration process, the operators are suggested to pay special attention to the software/hardware compatibility issues because such errors are difficult to detect with automated tools. It is interesting to note that the CloudMig was able to detect only half of the access control errors in RUBiS. This is because these errors include MYSQL privilege grant operations which are embedded in the application itself and the CloudMig configuration validation tool cannot intervene with the internal operations of MYSQL. Overall, CloudMig detected 55% of the errors in RUBiS migration as shown in Table 1.

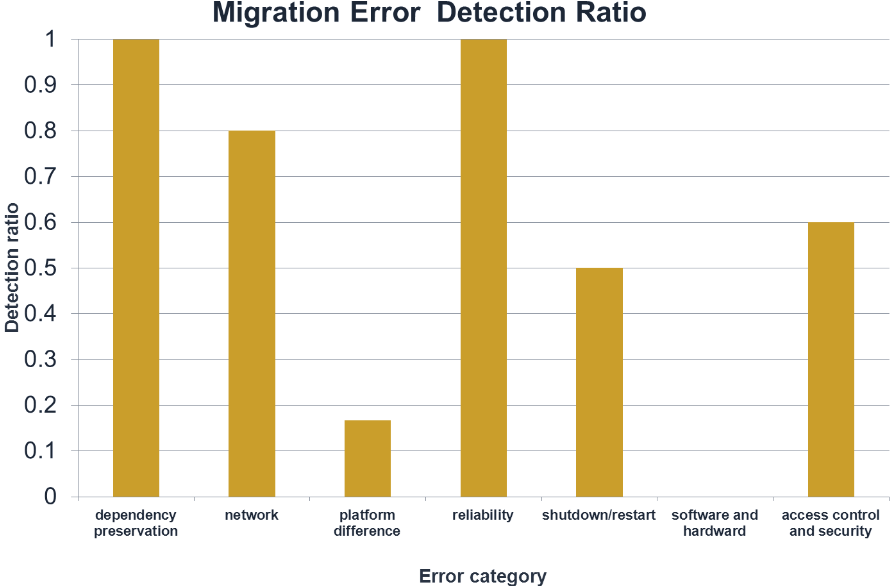


Figure 11. Overall migration error detection ratio

Figure 12. Overall migration error detection percentage. The legend lists the error types in the decreasing percentage order.

Figure 10 and Figure 11 show the number of detected errors and the error detection ratio of each error type summarized across the Hadoop migration case study and RUBiS migration case study respectively. Overall, CloudMig can detect all the dependency preservation and reliability errors and 80% of the network errors and 60% of the access control and security errors. In total, these four types of errors accounted for 74% of the total error occurrences. For shutdown/restart errors, CloudMig detected 50% of such errors and did not detect the software/hardware compatibility errors. This is because the application configuration data usually contains less information related with shutdown/restart operations or software/hardware compatibility constraints and this fact makes the configuration checking on these types of errors difficult without adding additional configuration policies. Figure 12 shows the percentage of error types in the total number of detected errors. One can see that 51% of the detected errors are dependency preservation errors, and 17% of the detected errors are network errors. Table 1 shows that totally across all the migrations, the error detection rate of CloudMig system is 70%.

Overall these experimental results show the efficacy of CloudMig in reducing the migration configuration errors, simpli-

fying the migration process and increasing the level of assurance of migration correctness.

7 Conclusion

We have discussed the system migration challenge faced by enterprises in migrating local data center applications to the Cloud platform. We analyze why such migration is a complicated and error-prone process and pointed out the limitations of the existing approaches to address this problem. Then we introduce the operator-based experimental study conducted over two representative systems (Hadoop and RUBiS) to investigate the error sources. From these experiments, we build the error classification model and analyze the demands for an semi-automated configuration management and migration validation system. Based on the operator study, we design the CloudMig system with two unique characteristics. First, we develop a continual query based configuration policy checking system, which facilitate operators to weave important configuration constraints into continual query policies and periodically run these policies to monitor the configuration changes and detect and alert the possible configuration constraints violations. In addition, CloudMig combines the continual query based policy checking system with the template based installation automation system, offering effective ways to help operators reduce the installation errors and increase the correctness assurance of application migration. Our experiments show that CloudMig can effectively detect a majority of the configuration errors in the migration process.

8 Aknowlegement

This work is partly sponsored by grants from NSF CISE NetSE program, CyberTrust program, Cross-cutting program and an IBM faculty award, an IBM SUR grant and a grant from Intel Research Council.

References

[1] Hadoop project. http://hadoop.apache.org/.
[2] RUBiS benchmark. http://rubis.ow2.org/.
[3] Amazon EC2. http://aws.amazon.com/ec2/, April 2011.
[4] Cloud Migration. http://www.opencrowd.com/services/migration.php, April 2011.
[5] Office Cloud. http://www.officetocloud.com, April 2011.
[6] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):610 –628, jul/aug 1999.
[7] R. A. Maxion and R. W. Reeder. Improving user-interface dependability through mitigation of human error. *Int. J. Hum.-Comput. Stud.*, 63:25–50, July 2005.
[8] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 04*, 2004.
[9] M. Vieira and H. Madeira. Recovery and performance balance of a cots dbms in the presence of operator faults. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 615–626, Washington, DC, USA, 2002. IEEE Computer Society.

Provenance for System Troubleshooting

Marc Chiarini
Harvard SEAS
chiarini@seas.harvard.edu

Abstract

System administrators use a variety of techniques to track down and repair (or avoid) problems that occur in the systems under their purview. Analyzing log files, cross-correlating events on different machines, establishing liveness and performance monitors, and automating configuration procedures are just a few of the approaches used to stave off entropy. These efforts are often stymied by the presence of hidden dependencies between components in a system (e.g., processes, pipes, files, etc). In this paper we argue that system-level provenance (metadata that records the history of files, pipes, processes and other system-level objects) can help expose these dependencies, giving system administrators a more complete picture of component interactions, thus easing the task of troubleshooting.

KEYWORDS: troubleshooting; diagnosis; dependencies; provenance; mental models.

1 Introduction

Most highly experienced system administrators can remember a time in their career when they were virtually clueless about the configuration of their systems. Whether learning on the job as a junior sysadmin or walking into a brand new infrastructure, nobody is ever handed a comprehensive guide to "the way things work around here." Instead, sysadmins must slowly develop a *mental model* of the systems in their care [6, 15]. They study existing documentation and Internet sources, solicit expert advice, explore component interactions, and much more. While this process is valuable in the long run, it is also time-consuming and error prone, and competes with the efficiency of whatever task is at hand (e.g., tracking down and fixing the root causes of problems). Additionally, mental models are developed on an as-needed basis and fail to account for hidden dependencies between system components, resulting in large gaps and inaccuracies.

This paper explores how system-level provenance can effectively expose hidden dependencies, improve mental models, and help improve the troubleshooting process for system administrators. Our goal is to build a provenance analysis engine that can automatically construct an accurate, queryable map of component interactions for single systems, networked sites, and beyond. Imagine arriving at your desk on a Monday morning and being able to explore what your site looks like based on provenance collected over the weekend.

2 Dependencies

Efficient troubleshooting requires mental models that are sufficiently accurate and complete to suggest proper courses of action. One part of a good mental model is a map of dependencies between the various components in a system. At a high level, *components* can be thought of as subsystems (e.g., the web subsystem depends upon the filesystem). At the lowest level of abstraction, components consist of programs and their individual configuration parameters. At this level, a good mental model maps how parameter changes affect a program's dependencies.

For the purposes of this research, we loosely define *dependency* as the relationship created when information flows from one component to another in order for the recipient of that information to function correctly. For example, when a process loads a library, functions necessary to the core behavior of the process are transmitted to it from a file. The process is *dependent* upon the library being loaded into some part of memory and being made accessible. Likewise, when Apache starts, it reads necessary parameters from an external source of information (e.g., `httpd.conf`). Furthermore, Apache depends upon

its runtime environment to properly specify the location of `httpd.conf`.

These are obvious examples of dependencies, but note that the way in which we have defined dependency requires a clear understanding of what it means for a component to function correctly. Formally, *functional correctness* is determined by behavior: every input produces correct output, where the output also comprises error conditions. Thus, if a process outputs "file not found" for some input, it may still be functioning correctly. But this definition is too strict for our purposes.

System administrators have a general sense of how components are supposed to behave, and they can usually determine when something is awry. For example, misconfiguration of one or more components is a frequent cause of "abnormal" behavior. Formally, a DBMS that is configured with a parameter that directs it to the wrong dataset will produce the correct behavior *for how it is configured*, i.e., it will still answer queries as directed, etc. But the admin will see unexpected outputs because the inputs were different than expected. This leads us to an imprecise definition of "functioning correctly" as "exhibiting expected behavior".

3 The PASS Project

Digital *provenance* is metadata that describes the ancestry or history of a digital object. In non-digital domains, such as art curation, provenance is often collected manually. But in the digital domain, we have the capability to record provenance automatically. The *provenance-aware storage system* (PASS) project [26] currently collects system-level provenance from inside a running kernel and builds a directed acyclic graph that describes ancestral relationships between files, pipes, and processes[1].

The provenance graph would be virtually useless without a way of extracting pertinent information. We have developed a query language for graph-structured data called PQL [14, 13], which is capable of expressing complex queries with transitive closures. PQL operates on a semi-structured data model that allows us to ask questions about ancestors and descendants as well as about paths and subgraphs.

Consider the case in which we want to find all outputs of the `sendmail` daemon. The following SPARQL[2] query produces the desired result:

```
SELECT ?output WHERE {
   progfile "/usr/sbin/sendmail" ?process .
   ?output output-of ?process
};
```

[1]This includes variables and other information about the environment in which they execute.

[2]PQL is similar to SPARQL [32], an SQL-like query language for RDF.

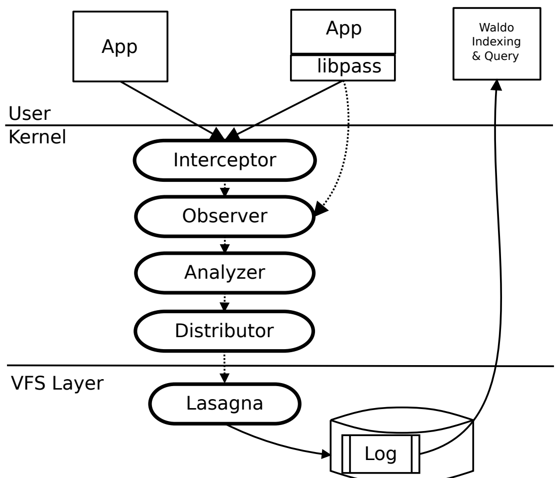


Figure 1: A diagram of the PASS Architecture.

In this example, *?output* and *?process* are variables. For every process that is an instantiation of `sendmail`, the query will return the process's output objects (e.g. files, pipes, processes, etc) in the variable *?output*. With PQL or a similar graph query language, we can issue simple queries such as the one in our example or complex queries such as "find all objects that result from the same (or similar) sequence of events", which is a path finding query.

If we think of files, pipes, and processes as system components between which information flows, then the provenance graph can be viewed as a graph of *potential dependencies*. Nodes of the graph represent components and edges represent a "may depend upon" relationship from one component to another. In practical terms, for a process P that reads from a file F, there exists a directed edge from the descendant P to the ancestor F. Likewise, if the same process writes to a pipe I, an edge from I to P will be generated in the graph. The graph describes only potential dependencies, because in the absence of code and dataflow analysis, we cannot be certain that any descendant depends upon its ancestors to *function correctly*.

3.1 PASS Architecture

Figure 1 shows the PASS architecture.[3] The *interceptor* is a set of system call hooks that extract arguments and other necessary information from kernel data structures, passing them to the *observer*. Currently, PASS intercepts `execve`, `fork`, `exit`, `read`, `readv`, `write`, `writev`, `mmap`, `open`, `pipe`, and the kernel operation `drop_inode`. These calls are sufficient to capture the

[3]The modified image and description of architecture are used with permission from the authors [26].

rich ancestry relationships between Linux files, pipes, and processes. In addition, applications can be compiled to use `libpass`, which allows us to send application-specific provenance directly to PASS.

This raw "proto-provenance" goes to the *observer*, which translates proto-provenance into provenance records. For example, when a process P reads a file A, the observer generates a record that includes the fact that P potentially depends upon A (i.e., a cross-reference to A). The first time an object is created, the observer assigns to it a unique *pnode* identifier. A pnode number is similar to an inode number except that it is never recycled, even after an object is destroyed. This allows us to maintain provenance for every object of interest that ever existed. Suppose that files A and B and process P have all been assigned pnodes. When P exits, its pnode must be maintained so that the transitive potential dependency of B upon A can be queried. The same logic holds for the case in which A is deleted.

The *analyzer* then processes the stream of provenance records to eliminate *duplicates* and *cyclic dependencies*. Duplicates occur when a provenance object is used as input multiple times in the same "session" by another object. For example, after the initial read of pipe I by process P, every further read creates a duplicate record until the pipe is closed. Yet a record of the initial read is all we require to posit a potential dependency.[4] In similar fashion, the provenance of a file F to which P has written multiple times will only contain a single record of the initial write.

Unless time travel is possible[5], it is impossible for a descendant object to affect its ancestor. This is why cycles in the provenance graph must be broken or avoided by the analyzer. PASS avoids cycles by versioning. At the time of its creation, each provenance object is assigned a version number of 0. New versions of an object will be assigned monotonically increasing numbers. If process P reads from file F, and later writes to that same file, the analyzer will avoid a cycle by versioning the file's provenance. If P reads the file again, the new record for this event will contain a cross-reference to Fv_1. That is to say that once F is written, further provenance will be collected only for subsequent new versions, and the provenance of Fv_0 will contain only whatever may have happened to F prior to the write and that didn't involve a cycle. The versioning algorithm works well on cycles of any length, involving any type of object at any version level.

The PASS system is not limited to collecting provenance from local storage. We have implemented extensions that enable provenance collection from NFS shares and Amazon's S3 service [3]. This capability is especially important to the multitude of organizations that have shifted their infrastructure into the cloud [27, 28].

Note that the interceptor is platform-specific by necessity, but that the observer and analyzer can be separated entirely from the operating system. The remaining components of the PASS architecture are not germane to the goals of this research. For a more complete description, we direct the reader to several prior works [25, 26].

[4] In general, this level of granularity imposes limitations on our ability to classify dependencies, i.e., we could keep the duplicates with a timestamp for more accurate resolution.

[5] There is now strong evidence to suggest that it is not [40]!

4 Troubleshooting

4.1 Related Work

In the past decade, there has been exciting research on improving failure diagnosis for system administrators. Some approaches use visualization to help operators rapidly detect and diagnose problems [36]. Others use event correlation in log-file analysis to identify extant and potential problems [1, 12, 17, 20, 34]. Wang et al. [37, 38] use comparisons of current system configurations against golden state configurations that have been generated via statistical analysis of machine populations. The HPC community has made significant strides in tracking down and diagnosing the root causes of failures in grids and clusters [2, 9, 31, 39]. Most of these approaches rely upon log analysis and can be extremely effective, especially in prescribed domains. However, log analysis may suffer from several drawbacks, including a lack of operational context (expected behavior); a "butterfly" effect on log messages that stem from small changes; corrupted messages; inconsistent log formats; and asymmetric log reports [29].

In the absence of formal documentation, sysadmins have few resources for determining the dependencies of a program. There exist tools that support static extraction of dependencies via analysis of package management repositories [18] and program images [35], but these have quite limited capabilities. For example, the former tool relies upon the correctness of package prerequisite information, and the latter tool only exposes compile-time dependencies.

Some tools [7, 33] are able to automatically construct operational dependency models by actively perturbing or probing live systems. Active perturbation involves performing multiple transactions or injecting "problems" outside of normal operation and tracking the affected components by observing likely execution paths. These methods are invasive, with the potential to cause unwanted load or unforeseen failures, and thus may be untenable in a production environment.

There are also many other approaches for exposing complex dependencies and causal relationships in dis-

tributed systems[4, 8, 11, 30], but their ability to document, present, and query the models they build is limited. This makes them ill-suited for improving mental models and for generalized system and site-wide troubleshooting.

Two research projects reflect well the philosophy we wish to propagate. PDA is a tool for automated problem determination developed at IBM [16]. The tool starts with high-level health indicators that trigger custom-built probes when something is awry. The probes are built manually via analysis of trouble-ticket corpora. Their use-case scenarios reveal that a large number of problems fall into several categories to which standard troubleshooting procedures can be applied and perhaps even automated. We are optimistic that these categories will also manifest in our provenance graphs.

We recently discovered a tool that is similar–both in concept and implementation–to the framework we propose in this paper, but more narrow in scope and no longer actively developed. BackTracker [19] is designed to analyze system intrusions by tracing chains of events from a detection point (e.g., a suspicious process) back through a dependency graph to likely points of entry. The goal is to document the attack vectors that expose unknown vulnerabilities. Similar to our approach, the graph is constructed by intercepting and recording the information in system calls. The authors also provide several security-specific methods by which to prioritize and filter large portions of the dependency graph to help the user along. The requirements for system troubleshooting are more general, thus our work may be viewed as an attempt to address a superset of the issues tackled by BackTracker.

Although one may assume that documentation is available for general-use tools, many organizations develop in-house solutions. When these solutions are intended for internal use only, there is little economic incentive to create polished user interfaces or comprehensive documentation; tools must simply be "good enough." As the number of internal libraries, scripts, and programs increases, making changes to the system becomes increasingly difficult. For example, deleting old libraries becomes virtually impossible when sysadmins have little knowledge of what programs utilize which libraries. The complexity of these poorly understood systems will continue to grow without bound as long as they are actively developed. Sysadmins in this situation would benefit greatly from a comprehensive and explorable graph of component dependencies.

4.2 A "Simple" Example

As suggested earlier, a clear and accurate system model is paramount to troubleshooting. Although sysadmins already troubleshoot in the absence of such models, their efforts have been significantly hindered by complexity. When something fails in a system, knowing where to look first is usually a "gimme". Under progressively greater pressure, knowing where to look second, third, fourth, and so on, requires experience and perseverance.

For example, in most UNIX distributions, the resolver, which sends DNS queries to translate names into IP addresses, loads its configuration from the file `/etc/resolv.conf`. Traditionally, this file was edited manually. In modern distributions such as Ubuntu, the file is now automatically generated and modified by the `NetworkManager` daemon. Various options for the network manager can be configured via GUI or the command line, but not resolver-specific options. Instead, if the host obtains its network configuration via DHCP, changes to `resolv.conf` are governed by the network manager's communication with the `dhclient` daemon, using D-Bus IPC[6]. The behavior of `dhclient` is in turn configured via the file `/etc/dhcp3/dhclient.conf`.

Given the dependencies just described, where does the system administrator look when she determines there is a problem with name resolution? The first place she may look is `resolv.conf`. Luckily for her, there is a comment in the file that states it has been automatically generated by the network manager. However, this is where the trail goes lukewarm. The manual page for the network manager says nothing about the resolver. Perhaps the sysadmin recalls that name resolution failures can be symptomatic of DHCP misconfiguration, leading her to check the dhclient manpage and subsequently `dhclient.conf`. She may find some useful information there, but she is hard pressed to discover that the network manager is modifying the resolver's configuration by talking to the DHCP client. Also, `dhclient.conf` may have been configured by an automated script. The trail goes cold until Google is consulted and a solution is discovered. But this is unsustainable as a standard procedure for troubleshooting; eventually, even Google is out of answers.

Using a provenance graph (Figure 2) and the right query types (or tools we build specifically for this purpose), our fearless administrator would more quickly discover the dependencies in our example. Let us walk through the troubleshooting session once more with the help of provenance. The graph has been trimmed and condensed for clarity, so the steps taken in an actual session may be more involved. Also, the following analysis suggests that we are able to collect provenance for remote sockets. This is not currently the case for PASS, but we are working on such a mechanism.

[6]The D-Bus implements inter-process communication (IPC) via Unix sockets, with each endpoint represented as an inode object and two file objects in the kernel.

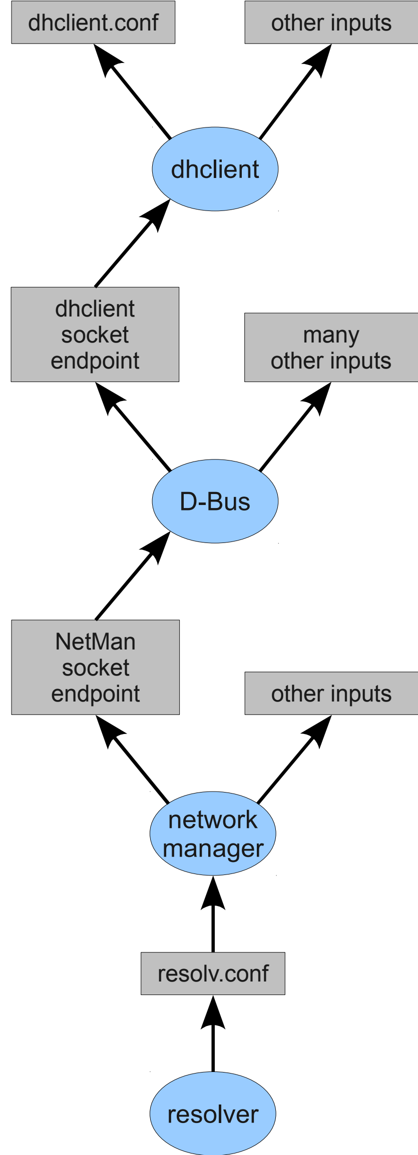


Figure 2: A partial provenance graph representing potential dependencies between components involved in Linux name resolution.

We may safely start at the network manager node (hereafter referred to as *netman*), since we already know the source of the generated `resolv.conf`. The ancestors of netman include a socket endpoint (a "special" file) and various other inputs, one of which will be a configuration file. We can probably safely exclude the configuration file, because there is nothing in netman's documentation about resolver options. But why has netman received information from a socket via the D-Bus? It has obviously communicated with another process. Here is where we run into a slight snag: D-Bus often has a plethora of socket endpoints as inputs (in addition to other inputs), so how can we determine the right ancestor? In many cases we may not be able to directly identify the most important ancestor but we can probably narrow down our choices.

One possibility involves checking timestamps of the provenance edges between objects of interest. In this case we could compare the timestamp of outputs to netman's socket endpoint with the timestamps of D-Bus inputs from any of its ancestors. We would discard D-Bus inputs that occurred after outputs to the socket as well as inputs that occur too long before outputs. Other technical solutions are also possible, including the recording of socket descriptors in provenance objects.

Once we have reasonably narrowed our choices, we will have to rely on experience to take us the rest of the way. Knowing that our machine receives network configuration parameters via DHCP will allow us to discard many other D-Bus ancestors, such as the audio, printing, and display subsystems. Once we reach the dhclient ancestor, we can determine which of its configuration options found in `dhclient.conf` are likely to be involved in name resolution.

The D-Bus example represents one of the worst-case scenarios in tracing root causes. The problem is twofold: at any given time, the number of ancestors and descendants of the daemon is usually very large, which results in an overwhelming path explosion; but the larger problem is that valuable provenance is hidden inside the D-Bus black box. For instance, if we had access to the internal dbus object name that identifies the connection between two clients, we could easily narrow our search to the real ancestors of the network manager. One way in which to accomplish this would be to create a provenance-aware version of D-Bus using the `libpass` library. This may be feasible for a small portion of particularly "opaque" system programs with many distinct inputs and outputs.

5 Ranking Dependencies

While the provenance of a process's outputs depends upon the process's inputs, the process itself is not necessarily dependent upon every input to function correctly. For example, the program `cat`, which reads the contents of an input stream, only depends upon three shared libraries to function correctly, yet a provenance graph includes edges to *every* distinct input object that `cat` opens. Though the absence of these inputs may cause a script to fail, none of them is essential to the core behavior of `cat`. This is why we have described the provenance graph as a graph of potential dependencies only.

A similar fact holds for many programs; almost every file (or other input) that is necessary for them to function properly is loaded with their image or shortly thereafter.

There are notable exceptions: programs such as Apache and PERL frequently load modules on-demand; daemons may reload their configuration files when a HUP signal is received, but will rarely reload a library; and shell scripts frequently defy all notions of predictability.

It would appear that the generated graph contains too much information for our purposes. Too many "unimportant" edges will make troubleshooting more difficult. Thus we need a way to limit the scope of our queries to those ancestral objects that are most likely to have contributed to the behavior or contents of a target descendant.

5.1 Statistical Approaches

There is a statistical approach that will help us rank the contribution to dependency made by individual edges, full paths, and ancestral subgraphs.

Consider that any given snapshot of a provenance graph represents events *as they actually happened*. Suppose that we look at a snapshot of the provenance graph generated between time t_1 and time t_2. We see an edge from the process `/usr/sbin/chpasswd` to the file `/etc/pam.conf`. We also see several edges leading from other objects to `chpasswd`. Let us examine what we know. We do not track information flow, so we do not know what `chpasswd` did with information that it read from `pam.conf`. We do not know if the process or its descendants would have functioned correctly if `pam.conf` was missing or contained different content. The graph only tells us that the *provenance* of `chpasswd` and its descendants *depended upon* `pam.conf` in its current state.

Let us assume that the process functioned correctly during the snapshot period. How do we assign a dependency rank to edges in the graph? One way might be to take multiple snapshots at equally spaced intervals and count the number of snapshots in which the edge of interest appears. A high count would indicate a higher likelihood of dependence. While this may seem reasonable, it will not work.

Recall that an object is uniquely identified by a pnode number, which remains the same through successive versions (and even unto death). Once a node becomes a part of the graph, it is never removed. Any edges connected to the node remain in the graph as well. Thus, there is no difference between snapshots except for the creation of nodes and edges, and increases in object versions.

The correct approach takes advantage of the logical separation between provenance objects. A process is the running instantiation of a particular program. As such, two separate invocations of a program (processes) will be assigned distinct pnodes and appear as distinct nodes in the graph. Figure 3 shows an example of this scenario.

Process P_1 has read file A, written file B and then

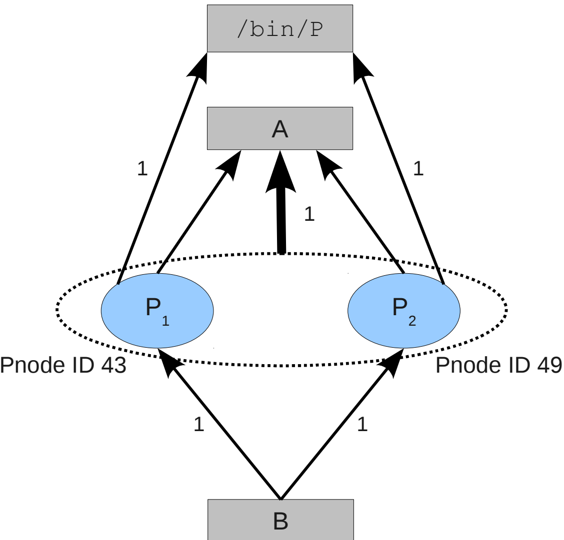


Figure 3: Processes with distinct pnodes. The program P (grouped processes) depends upon A with a ranking of 1 (thick edge).

terminated. Some time later, process P_2 takes the exact same actions. Notice that both processes have a provenance edge that points to the program executable `/bin/P`. For all processes that have a given executable as input, we can query whether or not they have a particular input (A in this case). If the same input object appears in the provenance of every process, then we declare that the current version of the program depends upon the input object with a ranking of 1.0. We denote this by grouping all such processes, drawing an edge from the group to the file, and labeling the edge with its rank. Alternatively, we can merge process nodes into a super-node to keep the graph clean.

There will be cases where only some instances of a program read from the same file. In these cases, only those instances are grouped and an edge is drawn to the file with a dependency rank given by

$$\frac{\texttt{\# of instances that read}}{\texttt{total \# of instances}}$$

We must not apply the same logic to rank the dependency of files upon programs. We do not know for certain what happens to the information that is read from a file by a process, e.g., whether it changes the behavior of the process. By contrast, a file always depends upon the process(es) that created and/or wrote to it. The reason is that a file is a passive object whose existence and content is governed by processes only. There is never any doubt that every bit of information in a file came from the process(es) that wrote to it.[7]

[7]Note that conceptually, if a process Q removes from a file all infor-

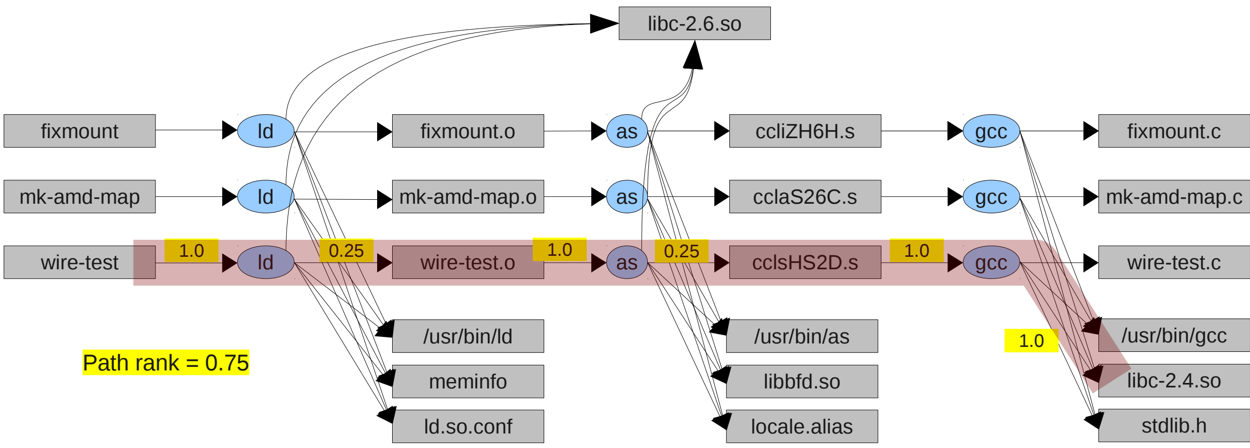


Figure 4: Dependency ranking for the path from `wire-test` to `libc-2.4.so`. The graph represents a mostly real provenance trace but the edge ranks are for demonstration only. If the executable really compiled, it would be difficult to identify the discrepancy between the two versions of libc.

Unnamed pipes are also passive objects that depend with certainty upon the process(es) that create them. Like processes, every new pipe is identified by a unique pnode number, and there always exists an edge to the process that created it. It might seem strange to claim that a pipe depends upon the processes that write to it, especially since we think of it as a simple channel by which processes communicate. The information sent to a pipe is meant to be consumed by one or more processes during the pipe's (relatively fleeting) lifetime. Unlike a file, none of the information in a pipe persists after it is torn down. Nonetheless, except for the timeframe, a pipe is serving one of the same purposes as a file; it is an information conduit between two processes.

Since named pipes are implemented as device special files, they remain usable after the process that created them exits. Thus we can use the same grouping technique to identify all processes that read from the same pipe, i.e., special file. With unnamed pipes, we need to go a step further. They are only connected between two single processes. It does not make much sense to claim that a program reads from the same pipe on every invocation. But we can claim that one program (i.e., every process from the same executable) always receives information from another program via a pipe, which implies a high-ranking transitive dependency upon the writer by the reader. A good way to represent this is to draw a directed edge from the group of processes to the group of pipes.

Armed with this metric, we can rank the potential dependence of paths or ancestral subgraphs. A path ranking is the average rank of all edges in the path. Similarly, a subgraph ranking is the average rank of all edges in the subgraph. For example, Figure 4 shows the rank of the highlighted path from the `wire-test` executable back to the ancestral file `libc-2.4.so`. We have omitted process grouping for clarity. When we query for the candidates that are likely causes of root problems, our tools should suggest exploration of the highest ranking paths first (accounting for rank adjustments from rules, filters, etc).

There are three caveats regarding this approach. First, it is has yet to be empirically tested. But our knowledge of operating systems provides a solid foundation. Second, the approach requires a bootstrap period during which rankings may be heavily influenced by existing abnormal behavior. This has the potential to mislead system administrators during analyses. We must therefore provide the ability for admins to manually adjust rankings in the graph, either permanently or via a "what-if" mode in a query session. The last warning is that while we expect the accuracy of rankings to improve over time, a large number of abnormal events may throw certain subgraph rankings into chaos at any time. We might be able to mitigate this by having the provenance subsystem alert us to statistical changes that exceed a certain threshold.

5.2 Heuristics

Statistical methods (and others) will carry us a fair distance in compressing the query space. But there is no reason to exclude existing knowledge about dependencies or rules of thumb. We now present several observations that will help us improve our rankings and refine our queries even further:

mation written by another process *P*, we might want to say that the file no longer depends upon *P*. But we have no way of representing this at our current level of granularity.

- Our current rules assign a dependency rank to edges based upon how many instances of a program read from the same input object. Informally, this says that for an edge with a higher rank than another, there is a greater chance that the input object affects the program's behavior.

  We might make the assertion that all but the simplest of daemons will always attempt to open and read from their associated configuration files. But note that if a daemon accepts a parameter that prevents loading of config files or specifies a different config file than the default (as many do), its input edges may receive a much lower rank than expected.

  Whether the daemon is dependent upon a specific config file is usually *conditioned* upon its starting parameters. Fortunately, PASS includes provenance about the environment in which a program is started. If we observe that a daemon always opens the same configuration file in the presence of some starting parameter, then we will be able to rank dependencies for different instances of the daemon (e.g., when '-c' is provided, the daemon *always* loads config file *C*, but in the absence of '-c', the daemon always loads config file *A*.). That is to say that we will group processes as usual but the group edge rank will indicate how many instances of the program read from an input object *when started with a given parameter*.

- The first-order dependencies of many programs are known a priori, either via direct experience, documentation, or technical detail, e.g. statically-linked programs. We can assign a rank of 1 to the outbound edges of these programs automatically upon first invocation.

- Popular objects, as measured by descendant subgraph size, are less likely to be the singular cause of a problem. It is a reasonable assumption that if a popular object is the cause of a problem, descendants along more than one path would exhibit unexpected behavior. In the case that we are only seeing one or a few objects with unexpected behavior, we can have our query engine dynamically reduce the dependency ranking for paths or subgraphs that include popular ancestors. The triggers for such a reduction and the amount of rank reduction will need to be determined by experiment.

  Example: almost every program has `libc` as a core library. This means that almost every edge that points to the `libc` node will have a dependency rank of 1. But this node is uninteresting exactly because so many programs depend upon it. If `libc` is broken or missing, we are likely to know immediately.

- Edges to files residing in well-known configuration directories or files with well-known names can be labeled with a high rank when all other indicators are equal or nearly so.

  For example, if a program *P* opens a file called `logrotate.conf` in directory `/etc`, then we have two more pieces of evidence to support the assertion that *P* depends upon `logrotate.conf`. The weight of this evidence will need to be adjusted according to several factors, which is left for future work.

  Of course, we must also provide a means by which we can fix the dependency or non-dependency of an object upon another object. This allows us to correct edges in the graph for which our algorithm has failed. There may be a semi-automated way in which to do this, which is also left for future work.

- Edges to files residing in well-known log directories can be labeled with a low rank.

  For example, `/var/log/messages` is a file that is frequently written, but certain log viewing/analysis/filtering/aggregation tools, such as Splunk, will frequently read the file as well.[8] In many cases, the absence or corruption of a log will not affect the proper functioning of the reading process. But it is difficult to know how far such a failure might propagate.

- Edges to files residing in well-known temporary directories can be labeled with a low rank.

  By definition, programs should not rely upon any data stored in a temporary directory (e.g. `/tmp`). However, programs do sometimes use such directories to create temporary pipes or to communicate information to themselves in the near future. These kinds of dependencies will need to be reviewed.

- Edges to files that are created by and opened for reading and writing in short intervals and across multiple invocations by a single program may be safely labeled with a low rank.

  For example, applications such as Emacs create backup files during editing. While the user may rely upon such backups, Emacs does not require these files to function correctly.

- Files that are created/written by an editor like `vi` are not dependent upon `vi`. They are dependent upon

[8] Many of these tools avoid the local filesystem altogether by logging to a centralized host via the network. In this case, provenance would be captured using network service extensions to PASS.

the human being using `vi`. This is a dependency that we can capture because we record the (E)UID of every process. If the file is created or written via shell redirection, we can still capture the dependency based upon the shell owner.

- Files created or modified by a script *are* dependent upon the script and probably many of its ancestors. But the path must ultimately lead back to the process that generated the script, whether manual or automatic.

- Any troubleshooting tools we build can integrate the use of whitelist, blacklist, Bayesian, and other filters. These will give the user flexibility in their queries and will certainly encourage use of the tool for purposes other than troubleshooting.

Acting intelligently upon the given observations will reduce the size and density of the query space. Note that none of our algorithms or heuristics is modifying the graph. Edge rankings will be applied only at query time based upon specified rules and filters, and will be computed in a lazy fashion. We do not want to rank one million edges for a single query unless it is necessary. For example, if a filter limits the query space to files in a particular directory, we do not need to rank edges from or to files in other directories, nor unnamed pipes.

As an example of where filtering may fail, suppose we determine that a program is behaving abnormally. It has file *A* as input, amongst others. A conventional rule of thumb may lead us to filter based upon time; the program was working until a certain point in time, so it is reasonable to ask which process most recently wrote to *A* around that time. But this may not help us because at the granularity of our provenance, the information that was most recently written to *A* might not be the information that is causing a malfunction. It is possible that some previous write is causing a malfunction. Perhaps the program did not run during the period between the previous write and the most recent write. Thus the effect of the previous write to *A* did not manifest until the program was run again.

6 Under-specified Queries

Filters and rules will help, but they are not sufficient. Even if we assume that the graph contains only actual dependencies, we still need the ability to limit the scope of *under-specified queries*. Such a query has the potential to return a very large subgraph because it does not sufficiently constrain ancestral breadth and depth. For example, if we query on the full lineage of `/var/log/dmesg`, we are likely to see all ancestors going back to installation of the operating system. Depending upon the context, this may be unhelpful. The ability to specify queries precisely assumes the existence of an excellent mental model by which to navigate the provenance graph. As the graph expands, "surgical" queries demand a familiarity that is unsustainable without aid. Thus, our tool needs to be able to guess at good places to stop in the lineage of a target object.

Several researchers in our group are attempting to tackle this problem based upon ideas inspired by web search [23]. *Provrank* is an algorithm that judges the importance of objects based upon their *frequency* across all possible lineage queries. Objects with a high frequency appear in too many lineage queries. Thus, if some process appears in the query path of every descendant object of interest, it does not add any important information to a query result and represents a good cutoff point. Another metric – *frequency dissimilarity* – captures the relative frequency of an object. That is to say, it measures how often an object appears in query results that contain objects of the same kind (based upon some criteria). Thus, the `bash` shell will have a lower frequency dissimilarity in queries that ask for the lineage of `mkdir`, than in queries that ask for the lineage of a random user document (i.e., the `bash` node would be a good cutoff point for queries about user documents).

Further work is required in this area to help sysadmins semi-automatically constrain their queries.

7 Building Tools

With a few decent algorithms under our belt, a gaggle of heuristics, and a good working knowledge of operating systems, what capabilities do we want for our tools and their interfaces?

In our resolver example, we guided the reader through a troubleshooting session that uses the provenance graph. Although based on a real use case, the example was directed and abbreviated for clarity. In a real session, a sysadmin would need a guide as well; something to improve their chances of diagnosing the problem.

Ideally, our tool must be able to present a relatively small group of root-cause candidates. But we are also helping admins build mental models. We expect to introduce several interfaces that leverage web technologies as well as the familiar command-line interface for conventional programmatic control.

Since PQL is the primary method of querying the provenance graph, we also plan to introduce a set of predefined query classes that will help users learn how to construct and refine more complex queries. A graphical tool is in the works that will enable the construction of queries via example as well.

Finally, integration is paramount. The user must be able to build the toolchain with relative ease and con-

nect it to existing monitoring and troubleshooting frameworks. For example, as problems are solved, relevant snippets of the graph and associated queries can be entered into a trouble ticket system and reviewed in subsequent incidents that exhibit similar symptoms.

7.1 Visualization

Provenance graphs can grow to enormous proportions, which tends to work against building robust mental models. Visualization can dramatically improve the ability for users to absorb and understand complex structures. As such, it is one of the most important aids to provenance analysis.

We have already built a tool called Orbiter [22] that can, among other capabilities, display provenance graphs with adjustable magnification, perform rudimentary filtering (e.g., degree, object type, timestamp, etc) and querying of ancestors and descendants, and summarize subgraphs at customized levels of granularity. We plan to extend Orbiter's capabilities with query subgraph highlighting, regular expression filters, process grouping, annotations, and programmable views. We will encourage system administrators to describe the most useful aspects of the tool, as well as their thoughts on whether and how to eliminate or improve its failings.

8 Future Work

The current implementation of PASS examines provenance as expressed only via pipes, shared memory (`mmap`), process environments, and the filesystem. Unfortunately, more sources of provenance (and potential dependencies) are expressed via other information vectors, e.g., signals, sockets, message queues, shared memory, semaphores, and exit codes. As a result, provenance graphs generated by our implementation are not comprehensive. We believe that analysis of network I/O will prove to be a powerful technique. By tracking socket pairs, we can identify dependencies that span physical machines. For example, a network-aware approach would be able to identify dependencies between a web server and a DNS server. Expanding the collection and analysis phases in this way will require considerable effort.

Another drawback in our current implementation is the inability to collect provenance from root volumes or to aggregate provenance from multiple disparate volumes. We are working to address these shortcomings by building a new collection platform [21] based in the Xen hypervisor [5] that obtains provenance directly from system calls inside of guest VMs. We expect this reorientation to yield new benefits, which include supporting a better case for adoption than a patched Linux kernel.

There are many other technologies that might be employed to help build and answer domain-specific troubleshooting queries, including further analysis of graph structure, more advanced statistical techniques, and a community-based query database. We also plan to incorporate ideas from machine-learning, not only to help conduct semi-automatic analyses of provenance graphs and provide better dependency rankings, but to augment graphs with information gleaned from interactions of system administrators with our tools [10, 24].

9 Conclusions

In our introduction, we made the claim that complete and accurate mental models are necessary to most tasks performed by system administrators, including troubleshooting and maintenance. As such, any tool that aids in the timely development of accurate mental models will be of great benefit to sysadmins at both the junior and senior level.

In this paper, we have explored the idea that analysis of provenance graphs can aid system administrators in troubleshooting problems that involve complex hidden dependencies. We are confident that if system administrators are amenable to automatic provenance collection, then this idea will emerge as an effective utility in everyday system administration.

10 Acknowledgments

The author would like to thank several members of the PASS group at Harvard University SEAS for their comments and insights on this research: Uri Braun, Peter Macko, Daniel Margo, and Margo Seltzer. He would also like to thank his LISA shepherd, Nicole Forsgren Velasquez, and the LISA program committee.

11 Availability

A working prototype is not yet available. However, readers are encouraged to periodically check the website below for news and updates.

`http://www.eecs.harvard.edu/syrah/pass/`

References

[1] Splunk. Web, June 2011. http://www.splunk.com/.

[2] AHARON, M., BARASH, G., COHEN, I., AND MORDECHAI, E. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I* (2009), Springer-Verlag, pp. 227–243.

[3] Amazon Simple Storage Service (Amazon S3). Web, June 2011. `http://aws.amazon.com/s3`.

[4] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review* (Aug. 2007), vol. 37, ACM, pp. 13–24.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[6] BARRETT, R., KANDOGAN, E., MAGLIO, P. P., HABER, E. M., TAKAYAMA, L. A., AND PRABAKER, M. Field studies of computer system administrators: analysis of system management tools and practices. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW)* (November 2004), J. D. Herbsleb and G. M. Olson, Eds., ACM, pp. 388–395.

[7] BROWN, A., KAR, G., AND KELLER, A. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management* (2001), pp. 377 –390.

[8] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the IEEE/IFIP 32nd International Conference on Dependable Systems and Networks (DSN)* (2002), IEEE Computer Society, pp. 595–604.

[9] CHUAH, E., KUO, S.-H., HIEW, P., TJHI, W. C., LEE, G., HAMMOND, J., MICHALEWICZ, M. T., HUNG, T., AND BROWNE, J. C. Diagnosing the root-causes of failures from cluster log files. In *Proceedings of the 2010 International Conference on High Performance Computing (HiPC)* (Singapore, Dec. 2010), pp. 1 –10.

[10] CUNNINGHAM, S. J., WITTEN, I. H., AND LITTIN, J. Applications of machine learning in information retrieval. *Annual Review of Information Science 34* (1999), 341–384.

[11] ENSEL, C. New approach for automated generation of service dependency models. In *Proceedings of the Second Latin American Network Operations and Management Symposium (LANOMS)* (Belo Horizonte, Brazil, Jan. 2001).

[12] HANSEN, S. E., AND ATKINS, E. T. Automated system monitoring and notification with swatch. In *Proceedings of the 7th USENIX Conference on System Administration* (1993), USENIX Association, pp. 145–152.

[13] HOLLAND, D. PQL language guide and reference. Web, June 2011. `http://www.eecs.harvard.edu/syrah/pql/docs/guide.pdf`.

[14] HOLLAND, D. A., BRAUN, U., MACLEAN, D., MUNISWAMY-REDDY, K., AND SELTZER, M. Choosing a data model and query language for provenance. In *Proceedings of the 2nd International Provenance and Annotation Workshop (IPAW)* (June 2008).

[15] HREBEC, D. G., AND STIBER, M. A survey of system administrator mental models and situation awareness. In *Proceedings of the 2001 ACM SIGCPR Conference on Computer Personnel Research* (2001), ACM, pp. 166–172.

[16] HUANG, H., JENNINGS, III, R., RUAN, Y., SAHOO, R., SAHU, S., AND SHAIKH, A. PDA: a tool for automated problem determination. In *Proceedings of the 21st Conference on Large Installation System Administration (LISA)* (2007), USENIX Association, pp. 153–166.

[17] HUANG, L., KE, X., WONG, K., AND MANKOVSKII, S. Symptom-based problem determination using log data abstraction. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)* (2010), ACM, pp. 313–326.

[18] KAR, G., KELLER, A., AND CALO, S. B. Managing application services over service provider networks: architecture and dependency analysis. In *Proceedings of the IEEE/IFIP 7th Network Operations and Management Symposium (NOMS)* (2000), J. W.-K. Hong and R. Weihmayer, Eds., IEEE, pp. 61–74.

[19] KING, S. T., AND CHEN, P. M. Backtracking intrusions. *ACM Transactions on Computer Systems 23*, 1 (Feb. 2005), 51–76.

[20] KRIZAK, P. Log analysis and event correlation using variable temporal event correlator (VTEC). In *Proceedings of the 24th International Conference on Large Installation System Administration (LISA)* (2010), USENIX Association, pp. 1–11.

[21] MACKO, P., CHIARINI, M., AND SELTZER, M. Collecting provenance in the xen hypervisor. In *Proceedings of the 3rd Workshop on the Theory and Application of Provenance (TaPP)* (June 2011), USENIX Association.

[22] MACKO, P., AND SELTZER, M. Provenance map orbiter: Interactive exploration of large provenance graphs. In *Proceedings of the 3rd Workshop on the Theory and Practice of Provenance (TaPP)* (June 2011), USENIX Association.

[23] MARGO, D., MACKO, P., AND SELTZER, M. Constraining provenance queries. NEDB Poster Session, January 2011. Presented at poster session of New England Database Summit 2011.

[24] MARGO, D., AND SMOGOR, R. Using provenance to extract semantic file attributes. In *Proceedings of the 2nd Workshop on the Theory and Practice of Provenance (TaPP)* (2010), USENIX Association.

[25] MUNISWAMY-REDDY, K. *Foundations for Provenance-Aware Systems*. Dissertation, Harvard, Mar. 2010.

[26] MUNISWAMY-REDDY, K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference ATC* (2009).

[27] MUNISWAMY-REDDY, K., MACKO, P., AND SELTZER, M. Making a cloud Provenance-Aware. In *Proceedings of the 1st Workshop on the Theory and Practice of Provenance (TaPP)* (2009).

[28] MUNISWAMY-REDDY, K., MACKO, P., AND SELTZER, M. Provenance for the cloud. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies FAST* (2010), USENIX Association, pp. 197–210.

[29] OLINER, A., AND STEARLEY, J. What supercomputers say: A study of five system logs. In *Proceedings of the IEEE/IFIP 37th Annual International Conference on Dependable Systems and Networks (DSN)* (2007), IEEE Computer Society, pp. 575–584.

[30] OLINER, A. J., AND AIKEN, A. Online detection of Multi-Component interactions in production systems. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)* (June 2011).

[31] OLINER, A. J., AIKEN, A., AND STEARLEY, J. Alert detection in system logs. In *Proceedings of the IEEE International Conference on Data Mining* (2008), IEEE Computer Society, pp. 959–964.

[32] PRUD'HOMMEAUX, E., AND SEABORNE, A. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[33] RISH, I., BRODIE, M., ODINTSOVA, N., MA, S., AND GRABARNIK, G. Real-time problem determination in distributed systems using active probing. In *Proceedings of the IEEE/IFIP 11th Network Operations and Management Symposium (NOMS)* (2004), pp. 133–146.

[34] ROUILLARD, J. P. Real-time log file analysis using the simple event correlator (SEC). In *Proceedings of the 18th Conference on Large Installation System Administration (LISA)* (2004), USENIX, pp. 133–150.

[35] SUN, Y., AND COUCH, A. L. Global impact analysis of dynamic library dependencies. In *Proceedings of the 15th Conference on Large Installation System Administration (LISA)* (2001), USENIX, pp. 145–150.

[36] TAKADA, T., AND KOIKE, H. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *Proceedings of the 16th Conference on Large Installation System Administration (LISA)* (2002), USENIX, pp. 133–144.

[37] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the 6th Conference on Operating Systems Design & Implementation OSDI* (2004), pp. 245–258.

[38] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. Strider: A blackbox, state-based approach to change and configuration management and support. In *Proceedings of the 17th Conference on Large Installation System Administration (LISA)* (2003), USENIX, pp. 159–172.

[39] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles (SOSP)* (2009), ACM, pp. 117–132.

[40] ZHANG, S., CHEN, J. F., LIU, C., LOY, M. M. T., WONG, G. K. L., AND DU, S. Optical precursor of a single photon. *Physical Review Letters 106*, 24 (June 2011).

Debugging Makefiles with remake

Rocky Bernstein

1 Remake

Autotools[Fou09, Fou10a] is still very popular as a framework for configuring and building open-source software. Since it is a collection of smaller tools, such as *autoconf, automake, libtool,* and *m4,* debugging code that it generates can be difficult.

When I wrote my first POSIX shell debugger for *bash,* one of my initial goals was to be able to debug autotools *configure* scripts, and I was rather pleased when it worked. It required, however, writing a custom *bash* module to read the 20,000 lines of shell script into an array much faster than *bash* was able to. (This module has since been incorporated into *bash* as built-in function *readarray.*) It was only after completing this task that I realized a POSIX shell debugger was just one part of the bigger problem of debugging autotools script. Here, I describe the next step in that endeavor, adding debugging to GNU Make[Fou10b, Ber11]. We will see how to use *remake* and a POSIX shell debugger (the one for *bash*) together.

Makefiles have been around for quite a while, and over time, largely through the success of *automake*, they have gotten more complex. *Make* can be somewhat opaque, but after writing the debugger component of *remake,* I can usually solve *make* problems very quickly and easily.

In many programming languages, such as POSIX shell, Perl, Python, Ruby, and Lisp, type expressions or statements have interactive shells to see what happens when they run. Although GNU Make is every bit as dynamic as these other languages, currently there is no such interactive shell. But the debugger briefly described here can serve as a handy substitute.

The programming language Ruby has a really interesting *make* equivalent called *rake*. (If you are writing something from scratch, please consider using both Ruby and *rake*.) But systems administrators often find themselves using tools and code written by others, and much open-source software uses *make*, via *automake*. *Make* is so pervasive that the reference implementations of Ruby use *make* to build themselves.

In keeping with my philosophy of trying to use the smallest hammer that will do the job, this paper shows some of the smallest changes of my forked version of GNU Make. When used in conjunction with one of my POSIX shell debuggers, you can dynamically debug commands issued by GNU Make into the POSIX shell.

1.1 remake –tasks

A useful feature of Ruby's *rake* program is that there is an option to print a list of "tasks" that one can perform. Tasks include things such as building the software, installing it, and running the tests.

In *make* terminology, tasks are a subset of "files" or "targets." However in *make*, we have to distinguish those files which are just supposed to be there in the source code from those that somehow get created; and many of the files that get created represent intermediate steps along the way to producing something larger. I find it good practice to borrow ideas from related tools, and I have added the `--tasks` option from Ruby's *rake.* This handles files in this way: if a target has a command to build it, then it is probably "interesting"; conversely, if there are no commands to build a target, that is, it is only listed as a dependency, then it probably is not interesting—it is there only to support other targets, and when it changes, it triggers other targets to be remade. Also, if a rule is a default rule of *make,* then it is probably not interesting. This would include things like the pattern rules for compiling a C program or extracting something from an archive or source-control system. The same notion of "interesting" is used in debugger stepping.

Here is `remake --tasks` for a typical Makefile system, using the `Makefile` that comes with the GNU Make distribution.

```
$ remake --tasks
.c.o
.c.obj
.dep_segment
CTAGS
ChangeLog
...
NMakefile
README
...
dist
dist-all
dist-bzip2
...
upload-alpha
upload-ftp
```

When I first looked at the output and saw `README` in this list of targets that have commands associated with them, I thought there must be a mistake, because `README` is usually a distribution file. So I broke out the debugger to check what was going on. The answer will become clear below, when I describe how to investigate targets with the debugger.

Another piece of interesting information we learn from this output is that there is a way to make the `ChangeLog` file, presumably from version control, and a way to make just the bzip2 tarball, or upload the distribution to the alpha and FTP sites.

Additionally, targets can have a description added for them so that they appear when the `--tasks` option is given. A description must consist of only one line and begins with `#:`. Here is a `Makefile` tagged this way:

```
#: Build everything
all:
      perl Build --makefile_env_macros 1

#: Create distribution tarball
dist:
```

```
        perl Build --makefile_env_macros 1 dist

#: Build and install package
install:
        perl Build --makefile_env_macros 1 install

#: Create or update MANIFEST file
manifest:
        perl Build --makefile_env_macros 1 manifest

#: Create or update manual pages
manpages:
        perl Build --makefile_env_macros 1 manpages
```

When run with the `--tasks` option we get:

```
all           # Build everything
dist          # Create distribution tarball
install       # Build and install package
manifest # Create or update MANIFEST file
manpages # Create or update manual pages
```

1.2 remake –trace example

Here is a real-world example of tracing and then debugging.

I made a change to the GNU Make source code and I wanted to make a new distribution. I ran the usual command that does this:

```
$ make dist
NEWS not updated; not releasing
make: *** [distdir] Error 1
```

It is not clear how NEWS needs to be updated, but more information might be available by consulting the rules for the `distdir` target.

Now let us run *remake*:

```
$ remake dist
NEWS not updated; not releasing
Makefile:887: *** [distdir] Error 1

#0  distdir at /tmp/remake/Makefile:887
#1  dist at /tmp/remake/Makefile:1004
Command-line invocation:
  "remake dist"
```

This shows additional information: the line number inside the Makefile for target `distdir` (887), the target that got us to this one, and the command-line invocation.

With standard *make*, we could use the name of the target `distdir` and search inside `Makefile` for that, but frequently it might not be in the top-level Makefile. Instead, it might be in some recursive invocation or in a file included from the top-level Makefle. The traceback information and file names reduce the detective work needed.

Now let us run *remake* with tracing turned on:

```
$ remake --trace dist
GNU Make 3.82+dbg-0.7.dev
...
Reading makefiles...
Updating goal targets....
  /tmp/remake/Makefile:1004 File ‘dist’ does not exist.
    /tmp/remake/Makefile:887 File ‘distdir’ does not exist.
    /tmp/remake/Makefile:887 Must remake target ‘distdir’.
to be continued...
```

The indentation in the lines containing file name and line numbers gives target level nesting: target `distdir` was asked to be remade because it is a dependency of target `dist`. With this option, we show the dependency nesting as we build and traverse the tree. Without it, we show these only at the point of error, if there is one.

```
Invoking recipe from Makefile:888 to update target ‘distdir’.
##>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
case `sed 15q ./NEWS` in \
*"3.82+dbg-0.7.dev"*) : ;; \
*) \
  echo "NEWS not updated; not releasing" 1>&2; \
  exit 1;; \
esac
##<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
NEWS not updated; not releasing
Makefile:887: *** [distdir] Error 1

#0  distdir at /tmp/remake/Makefile:887
#1  dist at /tmp/remake/Makefile:1004
Command-line invocation:
  "remake -x dist"
```

So finally we get the commands that were run to get that message, indicating which check was performed.

Another difference between *remake* and standard GNU Make is that *remake* has lines of the form:

```
##>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
##<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

These lines serve to separate shell commands about to be run from the output that they produce.

In brief, the changes to GNU Make to improve tracing and error reporting are:

- The Makefile file name and the line inside this file are reported when referring to a target
- On error:
  - a stack of relevant targets is shown, again with their locations
  - the command invocation used to run *make* is shown
  - an option allows for entering the debugger on error
- Shell input that is about to be run is separated from the output in running that shell code
- a `--tasks` option prints a list of interesting targets and any associated description line for each

1.3 remake --debugger example

The tracing described in the previous section will be enough for some purposes. But we can make the computer do more work to show us what is going on by using the built-in debugger.

Why does `README` appear when we run `rake --tasks` ? We can ask the debugger to describe the target `README`:

```
$ remake --debugger
GNU Make 3.82+dbg-0.7.dev
Reading makefiles...
Updating makefiles....
-> (/tmp/remake/Makefile:477)
Makefile: Makefile.in config.status
remake<0> target README
README: README.template Makefile
#  Implicit rule search has not been done.
#  Implicit/static pattern stem: 'README'
#  Modification time never checked.
#  File has not been updated.
#  Commands not yet started.
# automatic
# @ := README
# automatic
...
# < := README.template
# automatic
...
#  commands to execute (from 'Makefile', line 1329):
rm -f $@
sed -e 's@%VERSION%@$(VERSION)@g' \
    -e 's@%PACKAGE%@$(PACKAGE)@g' \
    $< > $@
chmod a-w $@
remake<1>
```

The file `README` is created from `README.template`. In the commands section, there are a number of expanded variables such as `$@` and `$<`. Earlier though, the values of the automatic variables `@` and < are shown; here they are `README` and `README.template` respectively. If, however, we want *remake* to do the expansion when showing the commands, there is an option to the `target` command for that:

```
remake<1> target README expand
README:
#  commands to execute (from 'Makefile', line 1329):
rm -f README
sed -e 's@%VERSION%@3.82+dbg-0.7.dev@g' \
    -e 's@%PACKAGE%@remake@g' \
    README.template > README
chmod a-w README
remake<1>
```

Although is is not immediately apparent, some expansion was done in showing the target and dependencies. Line 1328 in file `Makefile` looks like this:

```
$(TEMPLATES) : % : %.template Makefile
```

The debugger command `expand` can be used to get the expanded value of the variable `TEMPLATES`:

```
remake<2> expand TEMPLATES
Makefile:1319 (origin: makefile) TEMPLATES := README README.DOS ...
```

Now we return to tracking down what was happening when we tried to run `make dist`. Again we go into the debugger:

```
$ remake --debugger dist
GNU Make 3.82+dbg-0.7.dev
...
Reading makefiles...
Updating makefiles....
-> (/tmp/remake/Makefile:477)
Makefile: Makefile.in config.status
```

It appears that the first thing that is done is to check whether the Makefile itself is up to date. As before, we could list information from the target that we crashed on, `distdir`. However instead let us run until the target:

```
remake<0> continue distdir run
Breakpoint 1 on target distdir: file Makefile, line 887.
Updating goal targets....
  /tmp/remake/Makefile:1004 File 'dist' does not exist.
    /tmp/remake/Makefile:887 File 'distdir' does not exist.
.. (/tmp/remake/Makefile:887)
distdir
```

There are three interesting points in time when updating a target:

1. before checking dependencies of the target
2. after checking but before running commands to update the target
3. after runnning commands when the target update is finished

Adding `run` to the end of `continue distdir` causes us to stop after dependency checking.

The debugger first stopped before dependency checking, as shown by an icon, the two-character arrow `->`, so it lists the dependencies for the target. For the `Makefile` target, they were `Makefile.in` and `config.status`. After continuing, it next stops dependency checking, so dependencies of the target are not automatically shown, unless explicitly requested with `target` just as for the commands.

A common problem in designing this kind of tool is trying to figure out how to cut down the amount of information shown. We usually do not want a list of all dependencies for `distdir` here since that would include a list of all of the files in the distribution. With the `--tasks` option above, files without associated commands are dropped from the listing.

Another indication that the debugger stopped after dependency checking is that the two-character icon is `..` rather than `->`. I try to use analogous *gdb* commands when possible. Here, the *gdb*-like command `info program` makes the stopping place more explicit:

```
remake<3> info program
Starting directory '/tmp/remake'
Program invocation:
 remake  -X distdir
Recursion level: 0
Line 887 of "/tmp/remake/Makefile"
Program is stopped after rule-prerequisite checking.
```

At this point we can list the commands that are to be run next using the `target` command, which shows information regarding a target. We will use variables that have been set up by GNU Make when giving a target name. As we saw when listing variables for `README`, `@` is an automatically set variable containing the name of the current target. Since we have run to target `distdir`, `@` is set to that.

```
remake<1> target @ commands

distdir:
#  commands to execute (from 'Makefile', line 888):
@case 'sed 15q $(srcdir)/NEWS' in \
*"$(VERSION)"*) : ;; \
*) \
echo "NEWS not updated; not releasing" 1>\&2; \
exit 1;; \
esac
@list='$(MANS)'; if test -n "$$list"; then \
list='for p in $$list; do \
if test -f $$p; then d=; else d="$(srcdir)/"; fi; \
if test -f "$$d$$p"; then echo "$$d$$p"; else :; fi; done'; \
.. about 90 other lines.
```

Makefile commands can be confusing because there are two sources for variables: GNU Make variables and POSIX-shell variables. Here we see things like `$(VERSION)` which is a GNU make variable and `$$p` which is the POSIX-shell variable `$p`. An extra `$` needs to be added in the Makefile. We can ask the debugger to expand all of the Makefile variables, but instead, let us write this code out to a file using the `write` command:

```
remake<2> write
File "/tmp/distdir.sh" written.
```

We can use the *bash* debugger *bashdb* to debug the rest.

```
remake<3> quit
remake: That's all, folks...
$ bashdb /tmp/distdir.sh
bash debugger, bashdb, release 4.2-0.7
...
(/tmp/distdir.sh:4):
4: case 'sed 15q ./NEWS' in \
bashdb<3> step
(/tmp/distdir.sh:4):
4: case 'sed 15q ./NEWS' in \
sed 15q ./NEWS
```

If we do not know what `sed 15q ./NEWS` does, rather than look this up in a manual, we can let the debugger show us. The parentheses in the *bashdb* prompt mean that we are inside a subshell, the backtick part of `` `sed 15q ./NEWS` ``.

A useful command I added not too long ago to the debuggers is `eval` without any arguments. Here, it takes the line that is about to be run and runs it.

```
bashdb<(4)> eval
eval: sed 15q ./NEWS
Version 3.82+dbg-0.6
GNU make NEWS
 History of user-visible changes.
28 July 2010
...
```

One more step and we go to where we do not want to be:

```
bashdb<(5)> step
(/tmp/distdir.sh:7):
 7:   echo "NEWS not updated; not releasing" 1>&2; \
bashdb<6> list
 2:    #/tmp/remake/Makefile:887
 3:    #cd /tmp/remake
 4:    case 'sed 15q ./NEWS` in \
 5:    *"3.82+dbg-0.7.dev"*) : ;; \
 6:    *) \
 7: =>   echo "NEWS not updated; not releasing" 1>&2; \
 8:      exit 1;; \
 9:    esac
10:    @list='make.1'; if test -n "$list"; then \
11:      list=`for p in $list; do \
bashdb<7>
```

What is wrong is that we were looking for `3.82+dbg+0.7dev` inside the first 15 lines of the file NEWS and we did not find that.

The above example barely scratches the surface of what is available in both my GNU Make debugger and my POSIX shell debuggers. There is extensive help inside the debuggers and in the online manuals `http://bashdb.sourceforge.net/remake/remake.html/index.html` and `http://bashdb.sourceforge.net/bashdb.html`.

1.4 History and Acknowledgments

The idea for a GNU Make debugger came about after I had completed a debugger for *bash*[Ber09] and realized that there was much more to debugging distribution building in *autoconf* and *automake* scripts than just the *configure* script. So I first floated the idea in freshmeat forum[McC03]. A year later, in response to a challenge[Smi04], I wrote the first code without much trouble.

GNU Make already had a wealth of debugging information stored, so all that was needed was to keep track of a dependency stack and add calls to a REPL (read, eval, print loop) at appropriate times. Delving into the code to figure out the right times and places was the bulk of the hard work.

One suggestion is to display a tree or subtree of targets, possibly as a graph. Unfortunately, GNU Make does not save a tree of targets. Instead, it grows the branch it needs as it traverses targets and removes it afterwards. In order to provide debugging, I had to extend the code to save information from the current target back to the goal target.

So, some target actions can affect whether subsequent targets are up-to-date or not. To make things more complex, targets can be patterns that dynamically match the files created at run-time, and short of "building" the code, one can only give an approximation of existing dependencies.

I would like to thank Calyxa D. Tokay for her constant encouragement, and Stuart Frankel for turning my jumble of ideas into a slightly more coherent and well-organized paper. Tthe anonymous reviewers' comments were very helpful.

1.5 Availability

The home page for this project is `http://bashdb.sourceforge.net/remake/`. Download links for source code can be found there.

Yaroslav Halchenko has been providing Debian packages. The `git` source repository is at:
`https://github.com/rocky/remake`.

References

[Ber09] Rocky Bernstein. *Debugging with the Bash Debugger*, 4.2-0.8 edition, April 2009. Available from `http://bashdb.sourceforge.net/bashdbOutline.html`.

[Ber11] Rocky Bernstein. *Remake — GNU Make with comprehensible tracing and a debugger*, 3.82+dbg-0.7 edition, October 2011. Available from `http://bashdb.sourceforge.net/remake`.

[Fou09] Free Software Foundation. *GNU Automake*, 1.11.1 edition, July 2009. Available from `http://sources.redhat.com/automake/`.

[Fou10a] Free Software Foundation. *GNU Autoconf*, 2.6.8 edition, September 2010. Available from `http://www.gnu.org/software/autoconf/`.

[Fou10b] Free Software Foundation. *GNU Make*, 3.82 edition, July 2010. Available from `http://www.gnu.org/software/make/`.

[McC03] Andrew McCall. Stop the autoconf insanity! why we need a new build system., June 2003. Available from `http://freshmeat.net/articles/stop-the-autoconf-insanity-why-we-need-a-new-build-system`.

[Smi04] Paul D. Smith. *Re: Adding debugging to GNU make (Mailing lists are a disaster lately!)*, March 2004. Available from `https://lists.gnu.org/archive/html/make-alpha/2004-03/msg00001.html`.