Build Buddy For Fun And Profit

Dan Mills
Novell, Inc.

Abstract

We present a build and packaging system called Build Buddy. The system is comprised of a set of tools for building and maintaining software packages on multiple operating systems and architectures.

1 Introduction

Release engineering can be a complex and time-consuming task. Distribution vendors often employ multiple people to work full-time on building and packaging software. Ximian (Now a part of Novell) was faced with this problem as well, when releasing Ximian GNOME, a customized version of GNOME and other add-on applications. Ximian wanted to reach the widest possible audience, and that meant releasing on multiple platforms.

As any release engineer is aware of, releasing software on multiple platforms requires a good deal of effort. In addition to the complexities added by having different possibly incompatible versions of build dependencies (required libraries, tools, or other files), different distributions sometimes use different packaging systems, and even when they dont, they often have different packaging policies, which, if they are to be followed, add a new layer of requirements to be met.

Ximian took a stab at this problem, and created a collection of tools called Build Buddy to automate the process of building and packaging software. These tools allow release engineers to produce packages for multiple operating systems, verify package corectness, and submit packages to an external repository (such as a Red Carpet server). Building can be done at the push of a button, on a schedule, or continuously. Release engineers can also create customized build reports to be as plug-ins, or check job status through a variety of ways (from a Web UI to log files on disk).

2 Design And Implementation

The basic watchwords of build-buddy are "automation" and "reproducibility". Build Buddy's goal is to make it possible not only to build complex software packages (and test them) with a few simple commands, but to make it possible for developers creating the package to use the same process as the automatic world-building engines. Moreover, Build Buddy is designed to abstract the details of the underlying packaging system by encapsulating the build commands and packaging metadata in an XML file, which gets translated to the local packaging system of the current distribution. In this way, Build Buddy produces native packages which are installable by users without requiring any additional package management software. This is an important difference over most other cross-platform build environments: Build Buddy is designed to produce packages that tightly integrate into the systems they will ultimately be deployed to.

On a higher level, Build Buddy is responsible for setting up and enforcing protected build areas called jails, so a single machine can reliably build multiple copies or versions of products without stepping on anyone's toes by clobbering libraries or falling victim to version skew of installed dependencies.

On an even higher level, Build Buddy enables the build and packaging process to take place on a remote machine by providing an XML-RPC interface to a build daemon that waits for build requests, as well as a "master" scheduling daemon which is used to keep track of multiple build nodes and relay a build request to an appropriate node. There are web and command-line interfaces for submitting build job requests to the master, as well as the ability to mark a job to be re-run routinely as a snapshot.

2.1 Packaging System Abstraction

Build Buddy can produce native packages for the following packaging systems[1]:

- RPM[2] (Red Hat, SuSE, Mandrake, and others)
- DPKG[3] (Debian)
- SD[4] (HP-UX)

To achieve this, we use an XML description of the packaging metadata that will ultimately be converted to an RPM `spec` file, a DPKG `debian/rules` file, or an SD `psf` file. Although the format of all three vary quite a bit, much of the information is similar. For those cases when the information needs to be different per-platform, Build Buddy allows the release engineer to override chunks of the XML on a per-target basis.

2.1.1 Targetsets

Targetsets are a major feature of the Build Buddy XML package description file. They allow the writer to selectively apply chunks of XML to certain platforms, with a very simple syntax. There must be one targetset, called the "default" targetset, that matches everything:

```
<targetset>
    <filter>
        <i>.*</i>
    </filter>

    ...XML here...

</targetset>
```

After it, any number of additional targetset sections may be defined, matching any combination of targets:

```
<targetset>
    <filter>
        <i>suse-91-i586</i>
        <i>sles-.*-ppc</i>
        <i>fedora</i>
    </filter>

    ...XML here...

</targetset>
```

When the XML file is read, all matching targetsets are merged together, thus allowing the release engineer to place in the default targetset as much information as is common among different targets, and adding, removing, or changing those default on specific sets of operating systems or architectures.

[1] There is also IRIX Inst support, though largely unused. RPM platforms are the only ones supported for `bb_node`.

[2] http://rpm.org/

[3] http://debian.org/

[4] http://software.hp.com/products/SD_AT_HP/

2.1.2 Macros

Macros are a way of avoiding duplication of information. Our XML files make extensive use of these string replacement macros. Commonly, they are used to define parts of paths and other useful variables. For example, the `[[prefix]]` macro can be set-up to expand by default to `/usr` on various Red Hat Linux OSes, but to `/opt/gnome` on SuSE Linux OSes. This way, the command:

```
./configure --prefix=[[prefix]]
```

Will cause the software to be configured differently depending on the target being built on. Moreover, macros can reference other macros (as long as there are no recursive definitions), so it is possible to e.g., define a `[[configure]]` macro, which sets the prefix, sysconfdir, localstatedir, and other miscellaneous settings as desired on each target.

Macros are set in a global configuration file for all of Build Buddy, but they can be overridden on a per-module basis. When this is done, other macros that reference the overridden macro will use the new definition. So for example, if a module defines:

```
<macro id="prefix">/opt/myproduct</macro>
```

The `configure` macro will correctly configure the product to run from `/opt/myproduct`.

2.1.3 Project Organization

Build Buddy was designed to maintain a packages created from 3rd party sources. For this reason, the packaging metadata is kept separate from sources and patches. Each XML build file and its associated sources, patches, and any auxiliary files is called a "module".

In the general case, only a single XML configuration file per module is needed. Occasionally, however, other files must be provided. For this reason, Build Buddy projects are organized in CVS with one directory per module, and all of them are (usually) placed inside one CVS module.

2.2 Sources And Patches

Build Buddy uses a simple source repository, which can be set-up locally or remotely (using ssh) with minimal configuration. Files may be placed in the repository by using the `bb_submit` tool, which returns a repository handle that may be inserted in a `ximian-build.conf` file or used with the `bb_get` tool to retrieve the file. It is not possible, however, to remove a file from the repository. When an updated

version of a file with the same name exists, it is simply submitted again, and a new handle is produced–but the old handle is still valid.

The repository can be used to hold patches as well. These are treated equally by the repository, but they are used in a `<patch>` section in the XML packaging metadata, instead of a `<source>` section.

There exist several tools in Build Buddy to manage patches. While Build Buddy's patch management capabilities were not designed to supplant a version control system, it is possible to maintain source changes completely within Build Buddy. The `bb_regenerate` tool can generate a new patch by comparing a modified source tree to a pristine one. It can also regenerate a patch if it already exists. For example, if the sources have the following patches applied:

1. add-french-translation.patch-2
2. fix-autoconf.patch-1
3. fix-bug-51242.patch-1

And the second patch needs to be changed, it is possible to make those changes and ask `bb_regenerate` to create a new version of it, even if it partially overlaps with other patches. The new patch can then be tested and submitted to the repository (which will create the `fix-autoconf.patch-2` handle).

Build Buddy supports other source acquisition mechanisms in addition to the simple repository. HTTP and FTP behave similarly to the standard repository, but URLs are used instead of a repository handle.

It is also possible to use CVS or Subversion to acquire sources. In this case, Build Buddy will checkout the sources and attempt to create a distribution tarball. This command is configurable via the `<dist>` tagin the XML configuration file. If not specified, it will attempt to run `autogen.sh`, and `make dist`. Once a tarball has been created, it is used as the source for the rest of the build, and for the source packages. This practice ensures that a valid source package is created as part of the build process.

2.3 Basic Build Process

When building a single module, the following steps are generally taken (see fig. 1):

- Parse the XML build config file.
- Acquire all sources as defined.
- Produce local build/packaging files (e.g., RPM .spec).

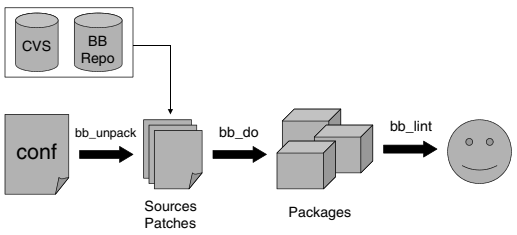

Figure 1: The basic build and packaging process

- Build and install to a temporary location.
- Package the built files.
- Test the resulting packages.

Build Buddy executes as many steps as possible from within the packaging system's environment. For example, on RPM systems, the build commands are executed through RPM. This helps ensure the source package will be valid and usable by third parties. However, when the packaging system does not support a certain step, Build Buddy performs it directly. For example, the SD system does not have a concept of a source package and simply expects binaries as its input files. In this case, Build Buddy emulates the RPM behavior by executing the build commands directly, and creating an additional package with sources.

2.4 Verifying Package Correctness

Even the best of packagers is bound to make a mistake sometime. To catch as many errors as we can before letting a package out the door, Build Buddy has a tool called `bb_lint`, which can be thought of as a unit testing framework, but restricted to packaging policy/metadata. It is possible to easily create new tests to be run either globally or for a particular module, or to disable a check for a particular module. Examples of existing tests include:

- Whether all files the module installs have been packaged.
- Whether there are any files in the package that are not owned by a system user (such as 'root', etc).

- Whether all packages have a "group" defined.

Some lint tests are designed to be run before the package has been built, such as the "group" check. Others, require the packages to be created, such as the test for non-packaged files. Tests can be flagged as errors, or warnings.

2.5 Building Large Products

In addition to the ability to build a single module on many operating systems, Build Buddy can also help manage the complexities of a large product with inter-dependent modules. It is possible to create another XML file called a `product` file, which describes the build-dependencies of all the modules involved. Then, one can use the `bb_build` tool to perform arbitrary operations (operations can be defined via plug-ins) on any subset of the module graph. For example, given the modules:

- `glib`
- `gtk+` (depends on `glib`)
- `nautilus` (depends on `gtk+`)
- `tcpdump`

It is possible to request a rebuild of `glib` and everything that has a dependency on it (`gtk+` and `nautilus`, but not `tcpdump`).

Due to the flexible operations, however, it is also possible to perform many other tasks that span multiple modules through `bb_build`. Examples include creating a HP-UX PSF file for a Bundle (a collection of products), running only specific portions of the build process (such as `bb_lint` by itself), or custom operations, such as unpacking a module and determining if it contains a certain source file or not.

Custom operations do require some programming and knowledge of certain Build Buddy data structures, but care has been taken to keep them easy to create and deploy. It is also possible to create a 'task', a sequence of operations that can then be conveniently invoked by a single name. For example, the default operation `bb_build` runs is 'build', which is a task that unpacks, builds, runs `bb_lint`, and cleans up the built tree. Thus, to execute the default sequence and add something at the end, the user can specify "build,oper" as the operation.

Upon encountering an error, `bb_build` can stop all operations on the current module as well as modules that depend on it. This is the defautl behavior. The user can also specify more or less lenient behavior, that is, to attempt to build dependent modules, or to halt immediately on error.

`bb_build` allows all output (including the output of the build process itself) to be sent to a file. This can be combined with home-grown scripts to integrate Build Buddy into a build process without using the XML-RPC Build Buddy daemons.

2.6 Jails

Jails are extensively used in Build Buddy. A jail image is essentially a tarball of an entire distribution, plus some metadata. When a jail image is unpacked, the chroot system call is used to enter the jail, so the build node is protected to some degree from the build, and vice-versa. However, as anyone familiar with chroot is aware, chroot does not provide a great deal of insulation–a number of things are shared, such as the process table, etc. Still, as a poor man's VM, it is excellent.

Build Buddy keeps arbitrary XML metadata associated with the jail image and with unpacked jails. Metadata currently used include hints to the node to control automatic deletion of old jails, information about the jail's owner, information about mount points that Build Buddy will automatically mount (such as /proc), etc. Since the metadata is arbitrary and easily parseable, external tools can tag jails, for example, for archival.

When the remote interface (Web UI / XML-RPC) is being used, it is possible to specify via XPath queries certain values to be met in the to-be-used jail. This makes it possible, for example, to deploy a specialized jail to be used for certain build jobs.

2.7 XML-RPC Interface

The components described so far operate on one or more modules, on a single machine or jail. Build Buddy also includes a set of networking components designed to automate the build process and manage a "build farm". In this set-up there is one "master" and many build "nodes". Each node registers itself with the master, which keeps track of them, and serves as a scheduler to distribute build requests, as well as a centralized point to collect logging information.

The current XML-RPC interface does not allow for full `bb_build` usage remotely. Instead, build requests specify each module to be built, in order. There are, however, plans to improve on this point by extending the XML-RPC interface.

The structure of the Build Buddy networked daemons is as follows (see fig. 2):

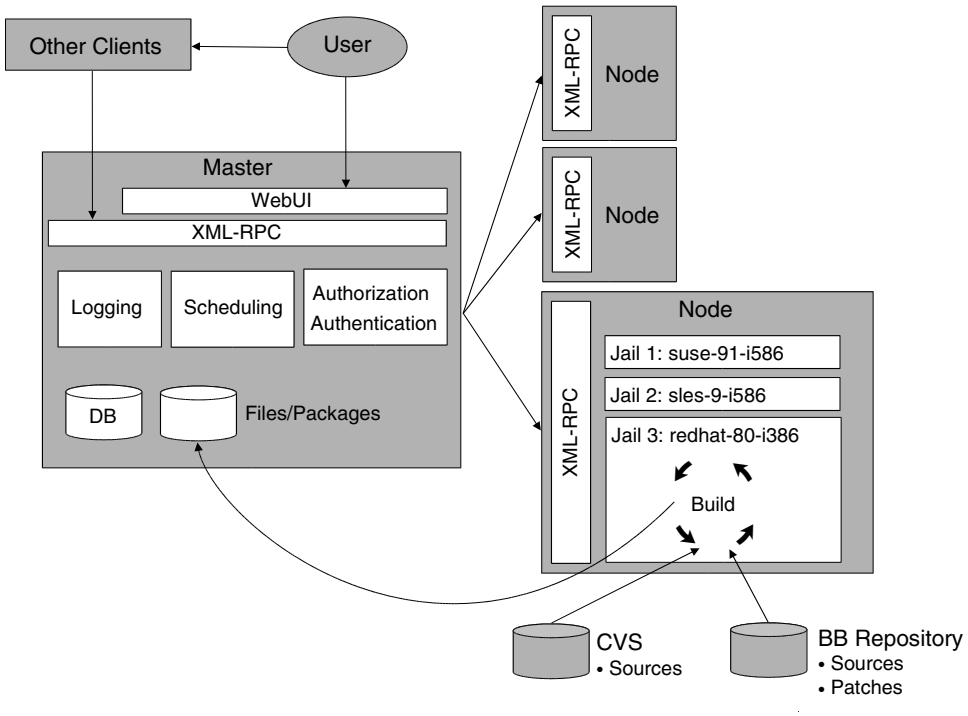

Figure 2: The networked build farm

The master is the main user-visible machine. Its purpose is to monitor the nodes, collect information about builds, and provide a single point of contact for the system. As such, it runs various daemons, including a web server, a database, and several Build Buddy services.

These services are:

- The logger is responsible for collecting all logging output from the jobs and storing it on-disk.
- The master is responsible for the scheduling of jobs, and for presenting a unified XML-RPC interface for all the available nodes.
- The authserver is responsible for user authentication, and also for authorizing or denying any requests from users.
- The snapshotter is an optional service to run saved jobs marked by users to be executed automatically on a regular basis.

The node is an individual machine which can build software upon request. Jails only need to run one background process, the `bb_node` daemon itself.

The actual build, which runs inside a jail, uses CVS to acquire the `ximian-build.conf` files. The sources to be built are defined in those files, and can point to external CVS, HTTP, FTP servers, or a Build Buddy file repository.

The build can also specify build dependencies, which are installed using the `Red Carpet` utility.

2.8 Remote Build Process

When a build request is submitted via the XML-RPC interface, the following steps take place:

1. The build master schedules the job to a node, taking into account the architecture requested, number of running jobs on each node, free HD space, etc.
2. The node searches for an available build jail for the requested os-version-arch (which Build Buddy calls a "target"). If none are available, a new one is unpacked.
3. Red Carpet is set-up inside the jail, to provide a method of installing build dependencies.
4. The XML build configuration files are checked out of CVS.

5. For each module to be built, these steps are performed:
   (a) Build dependencies are installed if needed.
   (b) The sources of the module are obtained from the tarball repository, or from CVS/Subversion.
   (c) `bb_lint` is run to check the XML metadata.
   (d) A `spec` file is generated for RPM. [5]
   (e) The software is built and packaged as per the `spec` file.
   (f) The resulting packages are synced over to the master (to avoid having to set-up a shared filesystem between the nodes and the master, rsync is used).
   (g) The resulting packages are tested for packaging errors with a second `bb_lint` run.
   (h) The resulting packages are installed inside the jail (which is necessary, since they can provide a build dependency for a later module).
6. Finally, if all went well, the packages are optionally submitted to a Red Carpet server, so that users can get at them, and so they can be used as build dependencies for other build jobs.

2.8.1 Logging

Throughout the entire process above, the status of the job, including up-to-the-minute process output logs can be seen via the Build Buddy Web UI. Nodes collect process output, and relay it to the master via a specialized XML-RPC daemon, which writes it to disk[6]. Reporting modules called "logstyles" are also available for snapshot jobs. Logstyles are event-based, and are highly customizable. Currently available logstyles include support for email reports (both text and HTML), and RSS.

2.8.2 Job Submission

There are three[7] interfaces to submit a build job to the master. One is a command-line interface called `bb_client`. The second is also a command-line utility, but it is designed to run on the master, for snapshot (recurrent) jobs, that is the `bb_snapshot` tool. The last method is the web interface. The web interface allows users to specify job information and save it to a database on the master. They can then submit the job, or they can mark it for snapshotting (which gets picked up by `bb_snapshot` when it runs).

[5] The node does not currently support Debian, HP-UX, or Solaris, even though the lower-level tools do.

[6] Previous Build Buddy versions used NFS to do this work, but it made the system harder to deploy, and was not more reliable than the current method.

[7] Actually, there are two more. One is an Emacs Lisp program to submit the XML config currently being edited for building, but it has not kept up with the latest XML-RPC interface changes. The other is a plugin for the Eclipse IDE to integrate with a Build Buddy installation at Novell Forge, which due to its Novell Forge-specific nature is not bundled with Build Buddy

3 Example Usage

The following is a working example of an XML build configuration file to build and package the 'Error' Perl module:

```
<?xml version="1.0" ?>
<!DOCTYPE module SYSTEM "helix-build.dtd">

<module>
    <targetset>
        <filter>
            <i>.*</i>
        </filter>

        <rcsid>$ Id: $</rcsid>
        <name>Error</name>

        <version>0.15</version>
        <rev>1</rev>
        <serial>1</serial>

        <psdata id="copyright">Artistic</psdata>
        <psdata id="url">http://www.cpan.org/</psdata>

        <source>
            <i>Error-0.15.tar.gz-1</i>
        </source>

        <build id="default">
            <prepare>[[perlprepare]]</prepare>
            <compile>[[perlmake]]</compile>
            <install>[[perlinstall]]</install>

            <package id="default">
                <name>perl-Error</name>
                <psdata id="group">Development/Perl</psdata>

                <files>
                    <i>[[perlmoddir]]</i>
                    <i>[[usrmandir]]/man*/*</i>
                </files>
                <docs>
                    <i>README</i>
                </docs>

                <description>
<h>Error extension for Perl5</h>
<p>This package provides a perl module.</p>
                </description>
            </package>
        </build>
    </targetset>
</module>
```

The above configuration file is placed in CVS, as described in the "Project Organization" section. To use the scripts directly, the conf is checked out on the build machine, and the various scripts (such as `bb_build`, `bb_do`, etc) are executed. The release en-

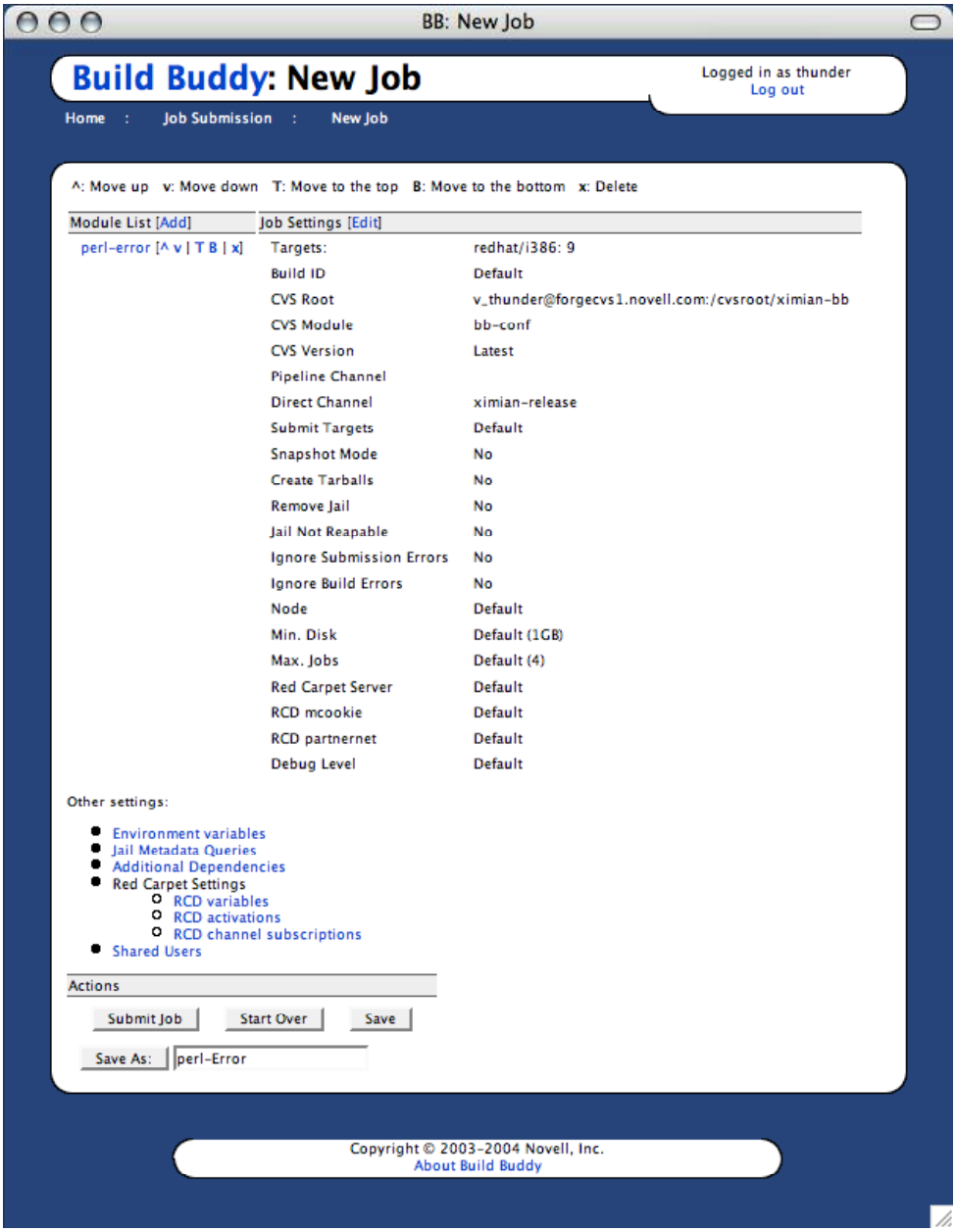

Figure 3: Job Submission

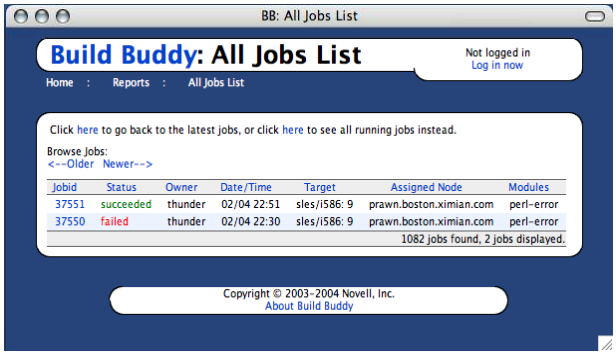

Figure 4: Recent Jobs (shortened list)

gineer can then use the resulting packages however is desired.

When using the WebUI, a new job is created for perl-Error (see fig. 3). The job includes information on where to check out the configuration file from, and which operating systems/platforms to build it on, as well as a Red Carpet server and channel to submit it to.

Since the configuration file specifies a tarball in the `<source>` tag (to be retrieved out of the Build Buddy file repository), this module will likely not be run as a snapshot. We can, however, update the configuration file stored in CVS at any time, increase the version as appropriate, and simply re-submit the job to build the new packages.

Once a job is submitted, it will appear under the "recent jobs" list (see fig. 4). The user can click on the job id to bring up job details (see fig. 5). From this page, it is possible to view individial log files, as well as download any generated packages, by clicking on them.

4 Future Plans

Although a great deal of effort has been spent on Build Buddy, there is still much that can be done. Aside from minor improvements in all areas, the biggest projects currently under consideration are a flexible way to keep track of a project's modules in the WebUI, and a redesign of the packaging system abstraction XML to allow for new platform support andd additional features.

4.1 WebUI Improvements

The WebUI is currently limited to a rigid definition of a build job, and has no definition of a project from which jobs can be derived. In reality, projects with multiple components will want to keep track of all the components without associating them to a "saved job", and be able to produce jobs based on the project data with less set-up than is currently needed.

To accomplish this, a logical container of modules can be created (the project), and the UI can be modified to allow build queries which allow the release engineer to make use of the bb_build command remotely.

4.2 Expanded Platform Support

Build Buddy's most unique feature is its ability to produce native packages on a wide range of platforms. Currently, this is restricted to UNIX-like operating systems, but there is no reason why this should continue to be the case. A research prototype exists that is able to build MSI packages on Microsoft Windows. This presents some interesting challenges, due to the differences between RPM, Dpkg, or SD, and Windows' MSI, but they still have enough in common that it is possible to use a single configuration file for all.

The Windows prototype is written in C#, and a new XML abstraction format was designed, both to add data only useful to MSI packaging, as well as to fix other long-standing problems or annoyances with the current XML format being used in production systems.

The MSI format is particularly different in its rich set of metadata for arbitrary files or folders. RPM and Dpkg do not generally require much additional metadata for files and folders. Flags for configuration files, documentation, permissions, or the like are sufficient in most cases, and several of these are mutually exclusive, making representation simpler.

The XML format implemented in the prototype has been designed to allow for arbitrary metadata for any resource (file, directory, or other) to be added at any later date. This is done by substituting the current file lists, which are simple strings, with a richer XML representation. At the same time, since string substitutions (Build Buddy macros) will not be useful for file lists, methods were added to produce the same results, while avoiding the need to be verbose.

4.3 Build Node Imaging

There are certain types of builds or platforms for which it would be best to completely convert a node to a different distribution, rather than use a Build Buddy jail. Some distributions make use of specific kernel features, for example, which prevent them from working properly in a chrooted environment. A solution to this problem is to use a technique generally

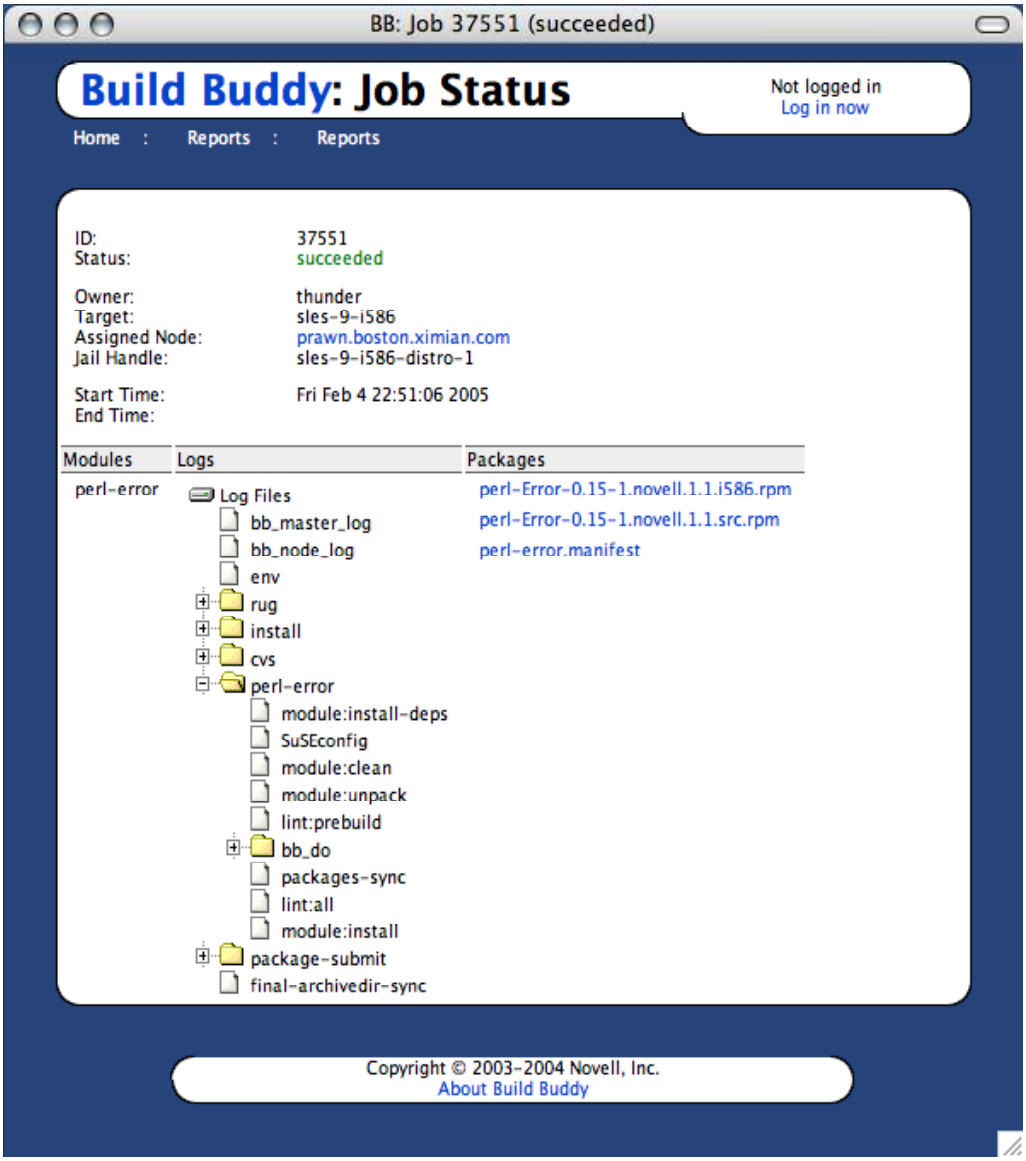

Figure 5: Job Details

known as "imaging". This technique uses disk images and network booting to achieve the equivalent of a full re-install of the operating system. Build Buddy could leverage this technology to augment or replace the current Jail system.

5 Related Work

Build Buddy is not the only project with the ability to build software and package it. One such tool is Maven (http://maven.apache.org/). Maven is a Java-based tool to help manage and build projects. It stores information about your project, from the names of developers to lists of dependencies, in an XML format, which it can then use in a variety of ways.

Another tool is SCons (http://www.scons.org/). SCons is a make, autoconf, and automake replacement, written in python. As with the tools it replaces, it is designed to precisely track the build process in detail, as well as build dependencies.

The biggest difference between Build Buddy and the above projects is that they concentrate on the build step, while Build Buddy's emphasis is actually on the packaging. It is possible to write scripts using any of the above tools to package software, but it would be a custom, per-project solution.

All of the above tools provide much greater detail and control over the specifics of the build process– that is, they solve problems such as: Which files need to be built before others? Exactly what commands need to be run to compile each file? Build Buddy, on the other hand, limits itself to invoking other tools to perform this work. In other words, although Build Buddy orchestrates the build process at a macro level, it invokes tools such as make, automake, or maven to achieve the specific build steps.

In this sense, Build Buddy is able to leverage the advantages of all of the above tools. Software authors are able to use Maven, SCons, make, etc. and simply invoke those tools from Build Buddy as needed. Build Buddy then focuses on the cross-distribution software packaging. Indeed, many software modules at Novell which are built using Build Buddy use make, autoconf, automake, or maven.

The disadvantage of this approach is that software authors who wish to use Build Buddy must use a combination of tools, rather than a single one, to perform the full build and packaging set of tasks. On the other hand, release engineers who integrate software from disparate places will find the Build Buddy's ability to adapt to any build tool highly desirable.

6 Summary

After more than four years of development, Build Buddy useful for a wide range of packaging tasks. By using all of its components, an experienced packager can develop and maintain large collections of packages for a number of distributions at the same time. At the same time, relatively novice packagers or developers, can use Build Buddy to make one-off packages, or maintain small sets of software with a minimum amount of training.

Build Buddy is licensed under the GNU GPL. Source code and additional documentation available at http://build-buddy.org/. We hope that you find it useful.

Acknowledgments

Many thanks to the original Build Buddy author, Mike Whitson, as well as Peter Teichman, Dave Camp, and other contributors. Thanks also to all Build Buddy users who have supported the project and helped shape it into what it is today. Last but not least, thanks to Nat Friedman and Miguel de Icaza, and other leaders at Ximian who had the foresight to invest in the project.

References

[BNS+00] Edward C. Bailey, Paul Nasrat, Matthias Saou, Ville Skyttä. Maximum RPM, 2000.

[JS98] Ian Jackson, Christian Schwarz. Debian Policy Manual, 1998.

[QR+01] Daniel Quinlan, Paul Russell, Filesystem Hierarchy Standard Group. Filesystem Hierarchy Standard 2.2, 2001.

[HP01] Hewlett-Packard Company. Software Distributor Administration Guide for HP-UX 11i, June 2001.

[ASF] Apache Software Foundation. Maven Project, http://maven.apache.org/.

[SF] The SCons Foundation. SCons Project, http://www.scons.org/.

[FSF] The Free Software Foundation. GNU Autoconf and GNU Automake, http://www.gnu.org/.