# Chukwa: A system for reliable large-scale log collection

Ariel Rabkin
asrabkin@cs.berkeley.edu
*UC Berkeley*

Randy Katz
randy@cs.berkeley.edu
*UC Berkeley*

## Abstract

Large Internet services companies like Google, Yahoo, and Facebook use the MapReduce programming model to process log data. MapReduce is designed to work on data stored in a distributed filesystem like Hadoop's HDFS. As a result, a number of log collection systems have been built to copy data into HDFS. These systems often lack a unified approach to failure handling, with errors being handled separately by each piece of the collection, transport and processing pipeline.

We argue for a unified approach, instead. We present a system, called Chukwa, that embodies this approach. Chukwa uses an end-to-end delivery model that can leverage local on-disk log files for reliability. This approach also eases integration with legacy systems. This architecture offers a choice of delivery models, making subsets of the collected data available promptly for clients that require it, while reliably storing a copy in HDFS. We demonstrate that our system works correctly on a 200-node testbed and can collect in excess of 200 MB/sec of log data. We supplement these measurements with a set of case studies describing real-world operational experience at several sites.

**Keywords**: logging, scale, research

## 1 Introduction

Almost every distributed service generates logging data. The rise of Cloud computing makes it easier than ever to deploy services across hundreds of nodes [4], with a corresponding increase in the quantity of logs and the difficulty of manual debugging. Automated log analysis is increasing the amount of information that can be extracted from logs, thus increasing their value [26, 30, 3, 16]. Hence, log collection and processing is increasingly important. Scalable data processing is challenging and so it is very desirable to leverage existing tools.

MapReduce is emerging as a standard tool for data-intensive processing of all kinds, including log file analysis [10, 29]. Tasks like indexing and aggregation fit naturally into the MapReduce paradigm. So do more sophisticated analyses, such as machine learning-based anomaly detection using console logs [30].

In this paper, we present Chukwa, a scalable system for collecting logs and other monitoring data and processing the data with MapReduce. Today, an administrator seeking to use MapReduce for system analysis would need to build a great deal of infrastructure to connect data sources with processing tools. Several sites have built such tools [23, 29], but each has been highly tailored to the specific context at hand. All have flawed failure recovery mechanisms, potentially leading to data loss. In contrast, Chukwa is designed to integrate cleanly with a wide variety of legacy systems and analysis applications and to offer strong reliability guarantees. It is available as open-source software and is currently in use at a number of sites, including Berkeley, Selective Media, and CBS Interactive. A previous publication described our initial goals and our prototype implementation [7]. In this paper, we describe Chukwa's design in more detail, present performance measurements, and describe real-world experiences.

### 1.1 Why distributed log collection is difficult

While MapReduce is a powerful and increasingly popular tool, there is a tension between its performance characteristics and those of many log collection workloads. One of the major design principles of MapReduce is to push computation to the node holding the associated data. This is accomplished by storing the input to a MapReduce job in a distributed filesystem such as the Google File System (GFS) [12], or its open-source counterpart, the Hadoop Distributed File System (HDFS). GFS and HDFS are user-level filesystems that do not implement POSIX semantics and that do not integrate with the OS filesystem layer. Both MapReduce and the un-

derlying filesystems are heavily optimized for the case of large files (measured in gigabytes) [6]. This means that applications must either be modified to write their logs to these filesystems, or else a separate process must copy logs into the filesystem.

This problem would be comparatively easy in a distributed filesystem that allowed multiple concurrent appends and where writes never failed. But such systems are quite difficult to build; no existing filesystem has both properties, and no system available in the open-source world has either. Support for single-writer non-concurrent appends has been in-progress in Hadoop for several years, despite implementation effort by a large population of paid full-time developers.

As a result, the implementation strategy adopted by the open-source world has been to implement this functionality in application code. In the standard approach, processes send their log messages across the network to a daemon, commonly called a collector, that serializes the updates and writes them to the filesystem. Several companies, such as Rapleaf, Rackspace, and Facebook [23, 29, 1], have built specialized log collection systems of this type.

These collection systems have largely treated log collection as just another network service. They expose a narrow interface to clients commonly using remote procedure call (RPC). The monitoring system is responsible for receiving and recording data and plays no role once data has been written to the distributed filesystem. While this separation of concerns is normally an attractive design style, we argue that it is the wrong approach for reliable monitoring of monitoring legacy systems.

A common task for a monitoring system is to collect data from legacy application log files on disk. Ideally, files on disk would be deleted once their contents have been stored durably by the monitoring system. But this is impossible without some way for the monitoring system to report back success or failure. In the event of a transitory failure, data may be buffered by the monitoring system for some time, meaning that a synchronous RPC model, with success or failure reported as soon as data is sent, is insufficient. This problem is perhaps less significant in organizations like Facebook or Google, where legacy code can be rewritten. But in smaller organizations, it looms large as a problem.

## 1.2 Our innovations

Our system, Chukwa, adopts a different architecture. Rather than expose a narrow interface to the monitoring system, we try to confine as much complexity as possible as close as possible to the application being monitored. This means that the interface between the system being monitored and the monitoring system is highly flexible and can be tailored to a particular context. It also means that the rest of the monitoring system can be simple and optimized for the common case. This enables substantial design simplification and good performance while offering superior reliability guarantees.

In Chukwa, data is collected by a dedicated agent process on each machine being monitored. This process can hold far more application-specific functionality than the simple network services offered by systems such as Scribe. As we show, this enables us to easily support a range of desirable features not found in alternative monitoring systems. Agents are responsible for three important tasks: producing metadata, handling failures, and integrating with existing data sources.

- Unlike other systems, Chukwa has a rich metadata model, meaning that semantically-meaningful subsets of data are processed together. This metadata is collected automatically and stored in parallel with data. This eases the development of parallel, scalable MapReduce analyses.
- We push failure handling and data cleaning to the endpoints of the monitoring system. Each agent is responsible for making sure that data is stored at least once. A MapReduce job removes duplicates. As a result, the interior of the collection system can be much simpler and can optimize for the common case where writes succeed.
- Last, we optimize for the case of log files on local disk. Such logs are common in many environments. Logs on disk are easy to create, easy to reason about, and robust to many failures. They are commonly produced by legacy systems. Chukwa demonstrates that these logs can also be used for low-cost failure recovery. From the point of view of Chuwka agents, data collection is asynchronous and need not be reliable. If a timer expires before data is stored durably, agents re-send using the on-disk log.

While Chukwa is optimized for logs on disk, it can handle many other monitoring tasks. Chukwa can collect a variety of system metrics and can receive data via a variety of network protocols, including `syslog`. Our reliability model encompasses these sources naturally and flexibly. Depending on user preferences, each data source can be buffered to disk pessimistically, buffered on error, or not buffered.

This work is timely for two reasons. The development of automated log analysis (such as [30, 3, 16] has made system logs much more useful. If logs are rarely consulted, then collecting them is a low priority. Now that system logs can be analyzed automatically and continuously, collecting them becomes a much higher priority.

The rise of Cloud computing makes it easier than ever to deploy services across hundreds of nodes [4], with a corresponding increase in the quantity of logs. At that scale, sophisticated storage and analysis tools like Hadoop become very desirable.

We begin, in the next section, by describing our design goals and assumptions and explaining why existing architectures do not adequately meet them. Section 3 describes our concrete implementation and Section 4 presents quantitative measurements. Section 5 discusses deployment experience. We describe related work in Section 6 and summarize our conclusions in Section 7.

## 2 Design Goals and Alternatives

Many monitoring and log collection systems have been built before Chukwa. In this section, we discuss our goals and why existing systems fail to meet them. These goals were based on design discussions at both Yahoo! and UC Berkeley and reflect real operational needs.

### 2.1 Supporting Production Use

We first list the core set of requirements needed to monitor production systems.

- The system must support a wide variety of data sources, not just log files. This is needed to collect system metrics and to cope with existing legacy systems that sometimes use other logging protocols, such as `syslog` [15].
- If the monitoring system fails, the system being monitored should continue working without interruption or slowdown.
- The system should scale to handle large numbers of clients and large aggregate data rates. Our target was to support 10,000 hosts and 30 MB/sec, matching the largest clusters currently in use at Yahoo [7].
- The system should impose low overhead. We have often heard 5% described as the largest fraction of system resources that administrators are comfortable devoting to monitoring. Lacking any more principled standard, we have adopted this as our target maximum resource utilization for the monitoring system.
- No matter how intense a bust of log writes, the resource consumption of the monitoring system should remain with its resource bounds.

Some log analysis jobs are very sensitive to missing data. In general, whenever the absence of a log message is significant to an analysis, losing even a small quantity of data can result in a badly wrong answer. For instance, Rackspace uses a MapReduce-based analysis of email logs to determine the precise path that mail is taking through their infrastructure [29]. If the log entry corresponding to a delivery is missing, the analysis will wrongly conclude that mail was lost. The machine-learning based log file analysis developed by Xu *et al.* [30] is another example of a loss-sensitive analysis. And of course, if web access logs are used for billing purposes, lost messages translate directly into lost revenue. To support these sorts of log analysis applications, we made reliable delivery a core goal for Chukwa.

Two of our goals conflict. A system cannot both offer reliable delivery in all circumstances while never having the system being monitored block while waiting for the monitoring system. Local storage is limited, meaning that if the monitoring system is unavailable for a long time, the system being monitored must either discard data or block. To reconcile these goals, we adopted the following reliability standard: if the machine originating the data stays does not fail permanently, data will eventually be delivered.

Making data available to MapReduce in less than a minute or two was not a goal. Chukwa was primarily designed to enable MapReduce processing of log data. Due to scheduling overheads, a Hadoop MapReduce job seldom executes in less than a minute. As a result, reducing data delivery latency below a minute offers limited benefit.

### 2.2 Why existing architectures are inadequate

Perhaps surprisingly, existing monitoring systems and architectures are inadequate to meet the goals listed above. The oldest and simplest form of logging is writing to local disk. Local disk writes are low-cost, and have predictable performance. Unfortunately, processing data scattered across local disks of a cluster is difficult. Doing so while processing data in-place will result in analysis workloads and production loads conflicting, which is often unacceptable in practice.

Doing processing on a separate analysis cluster requires some way of moving data from source to destination. A shared NFS mount and streaming data via `syslog` are two standard ways to do this. These two approaches make contrasting reliability-availability choices. If the network fails, an NFS write will fail, blocking the application. Syslog, built on UDP, will silently discard data.

That leaves writing data locally, either on failure or before attempting to copy it to HDFS. We discuss each in turn. Several systems, notably Scribe [1] attempt to

write data across the network, and buffer locally only on failure. The catch is that clients do not participate in failure recovery. Data loss will be fatal if a crash occurs after data has been handed to Scribe, and before that data has been stored durably. Likewise, data can be lost if a filesystem write returns success before data is serialized. (This can happen because HDFS buffers aggressively before flushing data to remote hosts.) As a result, pessimistic logging by the application is required to achieve high reliability.

For large files, copied periodically and all-at-once, this is simple to implement and works well. For streaming data, however, complexities emerge. There is an "impedance mismatch" between many logging workloads and the optimal performance envelope for scalable distributed MapReduce-friendly file systems like HDFS. Those file systems are designed for a small number of large files, written once and never updated. In contrast, large numbers of small log files updated sporadically are an important kind of monitoring data. Easing this gap requires consolidating logs from many machines into one file. Since HDFS lacks concurrent appends, this requires a separate process to do the merging. This increases the number of points at which failures can occur.

Chukwa responds to this reliability problem in an end-to-end manner, by pushing the retry logic as close to the data source as possible. Data is either stored in log files on local disks, or else in HDFS. No other copies are made, by default. Data transmission is only treated as successful once data from the one source has been successfully copied to the other. The technical challenge is two-fold. Reliability needs to be integrated with legacy applications that may be oblivious to the monitoring system. And this comparison must be performed efficiently and continuously at run-time.

Not all sources or uses of log data require the same degree of reliability. A site might decide that pessimistically recording all system metrics to disk is an unnecessary and wasteful degree of robustness. An additional design goal for us was to avoid imposing excessive costs for collecting this sort of ephemeral data.

### 2.3 A choice of delivery models

It became clear to us as we were developing Chukwa that in addition to reliability-sensitive applications, there is an another class of applications with quite different needs. It is sometimes desirable to use logs to drive an ongoing decision-making process, such as whether to send an alert to an administrator based on a critical error or whether to scale up or scale down a cloud service in response to load. These applications are perforce less sensitive to missing data, since they must work correctly even if the node that generated the missing data crashes.

| **Reliable delivery** | **Fast-path delivery** |
|---|---|
| Visible in minutes | Visible in seconds |
| Writes to HDFS | Writes to socket |
| Resends after crash | Does not resend |
| All data | User-specified filtering |
| Supports MapReduce | Stream processing |
| In order | No guarantees |

Table 1: The two delivery models offered by Chukwa

To support latency-sensitive applications, we offer an alternate "fast path" delivery model. This model was designed to impose minimal delays on data delivery. Data is sent via TCP, but we make no other concession to reliable delivery on this path. Applications using the fast path can compensate for missing data by inspecting the reliably-written copy on HDFS. Table 1 compares these two delivery models.

## 3 Architecture

In the previous section, we described our design goals. In this section, we describe our design and how it achieves these goals. Like the other systems of this type, we introduce auxiliary processes between the log data and the distributed filesystem. Unlike other systems, we split these processes into two classes. One set of processes, the *collectors*, are responsible for writing to HDFS and are entirely stateless. The other class, the *agents* run on each machine being monitored. All the state of the monitoring system is stored in agents, and is checkpointed regularly to disk, easing failure recovery. We describe each half of the system in turn. We then discuss our data model and the fault-tolerance approach it enables. Figure 1 depicts the overall architecture.

### 3.1 Agents

Recall that a major goal for Chukwa was to cleanly incorporate existing log files as well as interprocess communication protocols. The set of files or sockets being monitored will inevitably grow and shrink over time, as various processes start and finish. As a result, the agent process on each machine needs to be highly configurable.

Most monitoring systems today require data to be sent via a specific protocol. Both `syslogd` and Scribe [15, 1] are examples of such systems. Chukwa takes a different approach. In Chukwa, agents are not directly responsible for receiving data. Instead, they provide an execution environment for dynamically loadable and configurable modules called *adaptors*. These adaptors are responsible for reading data from the filesystem or directly from the application being monitored. The output

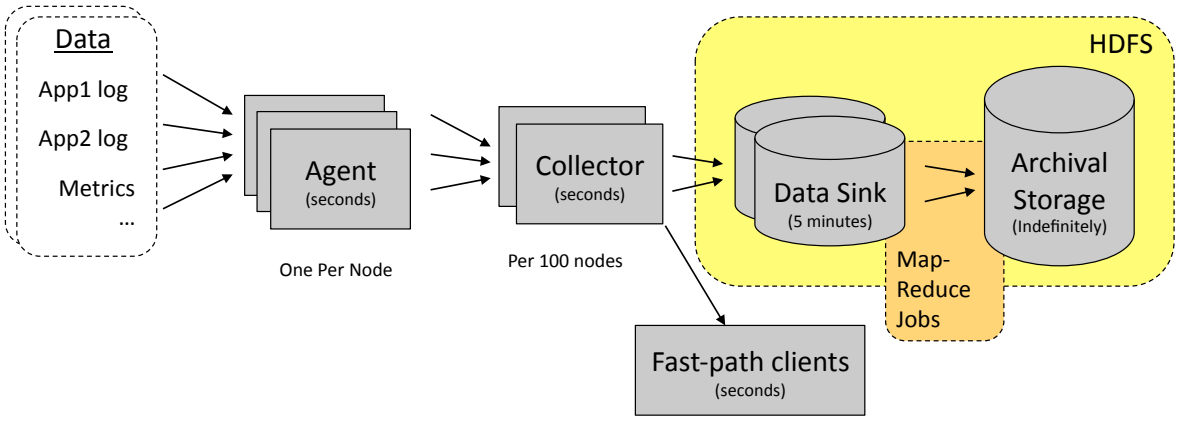


Figure 1: The flow of data through Chukwa, showing retention times at each stage.

from an adaptor is conceptually a stream of consecutive bytes. A single stream might correspond to a single file, or a set of repeated invocations of a Unix utility, or the set of packets received on a given socket. The stream abstraction is implemented by storing data as a sequence of *chunks*. Each chunk consists of some stream-level metadata (described below), plus an array of data bytes.

At present, we have adaptors for invoking Unix commands, for receiving UDP messages (including `syslog` messages), and, most importantly, for repeatedly "tailing" log files, sending any data written to the file since its last inspection. We also have an adaptor for scanning directories and starting a file tailing adaptor on any newly created files.

It is possible to compose or "nest" adaptors. For instance, we have an adaptor that buffers the output from another adaptor in memory and another for write-ahead logging. This sort of nesting allows us to decouple the challenges of buffering, storage, and retransmission from those of receiving data. This achieves our goal of allowing administrators to decide precisely the level of failure robustness required for each data stream.

The agent process is responsible for starting and stopping adaptors and for sending data across the network. Agents understand a simple line-oriented control protocol, designed to be be easy for both humans and programs to use. The protocol has commands for starting adaptors, stopping them, and querying their status. This allows external programs to reconfigure Chukwa to begin reading their logs.

Running all adaptors inside a single process helps administrators impose resource constraints, a requirement in production settings. Memory usage can be controlled by setting the JVM heap size. CPU usage can be controlled via `nice`. Bandwidth is also constrained by the agent process, which has a configurable maximum send rate. We use fixed-size queues inside the agent process, so if available bandwidth is exceeded or if the collectors are slow in responding, then back-pressure will throttle the adaptors inside the process [28].

The agent process periodically queries each adaptor for its status, and stores the answer in a checkpoint file. The checkpoint includes the amount of data from each adaptor that has been committed to the distributed filesystem. Each adaptor is responsible for recording enough additional state to be able to resume cleanly, without sending corrupted data to downstream recipients. Note that checkpoints include adaptor state, but not the underlying data. As a result, they are quite small – typically no more than a few hundred bytes per adaptor. This allows Chukwa to scale to many hundreds or thousands of files being monitored.

One challenge in using files for fault-tolerance is correctly handling log file rotation. Commonly, log files are renamed either on a fixed schedule, or when the reach a predetermined size. When this happens, data should still be sent and sent only once. In our architecture, correctly handling log file rotation is the responsibility of the adaptor. Different adaptors can be implemented with different strategies. Our default approach is as follows: If instructed to monitor log file `foo`, assume that any file starting with `foo.*` is a rotated version of `foo`. Use file modification dates to put rotated versions in the correct order. Store the last time at which data was successfully committed and the associated position in the file. This is enough information to resume correctly after a crash.

## 3.2 Collectors

We now turn to the next state of our architecture, the collectors. If each agent wrote directly to HDFS, this would result in a large number of small files. Instead, Chukwa

uses the increasingly-common collector technique mentioned in the introduction, where a single process multiplexes the data coming from a large number of agents.

Each collector writes the data it receives to a single output file, in the so-called "data sink" directory. This reduces the number of files generated from one per machine or adaptor per unit time to a handful per cluster. In a sense, collectors exist to ease the "impedance mismatch" between large numbers of low-rate sources and a filesystem that is optimized for a small number of high-rate writers. Collectors periodically close their output files, rename the files to mark them available for processing, and begin writing a new file. We refer to this as "file rotation." A MapReduce job periodically compacts the files in the sink and merges them into the archive of collected log data.

Chukwa differs in several ways from most other systems that employ the collector design technique. We do not make any attempt to achieve reliability at the collector. Instead, we rely on an end-to-end protocol, discussed in the next section. Nor do Chukwa agents dynamically load-balance across collectors. Instead, they try collectors at random until one appears to be working and then use that collector exclusively until they receive errors, at which point they fail-over to a new one. The benefit of this approach is that it bounds the number of agents that will be affected if a collector fails before flushing data to the filesystem. This avoids a scaling problem that would otherwise occur where every agent is forced to respond to the failure of any collector. One drawback is that collectors may be unevenly loaded. This has not posed any problems in practice since in a typical deployment the collectors are far from saturated. With a collector on every HDFS node, we have found that the underlying filesystem saturates well before the collectors do.

To correctly handle overload situations, agents do not keep retrying indefinitely. If writes to a collector fail, that collector is marked as "bad", and the agent will wait for a configurable period before trying to write to it again. Thus, if all collectors are overloaded, an agent will try each, fail on each, and then wait for several minutes before trying again.

Collectors are responsible for supporting our "fast path" delivery model. To receive data using this model, clients connect to a collector, and specify a set of regular expressions matching data of interest. (These regular expressions can be used to match either content or the Chukwa metadata, discussed in the next subsection.) Whenever a chunk of data arrives matching these filters, it is sent via a TCP socket to the requesting process in addition to being written to HDFS. To get full coverage, a client needs to connect to every collector. As we will show in the next section, a modest number of collectors are sufficient for the logging needs of large datacenter services. Hence, "every collector" is often only a handful.

Filtering data at collectors has a number of advantages. In the environments we have seen, collectors are IO-bound, not CPU-bound, meaning that CPU resources are available for the pattern matching. Moreover, collectors are stateless, meaning that it is straightforward to spread out this matching across more machines, if need be, by simply adding more collectors.

The fast path makes few reliability promises. Data can be duplicated, if an agent detects a collector failure and resends. Data can be lost, if the collector or the data recipient fails. In some failure scenarios, data can be received out of order. While data is normally delivered to clients as soon as it is received by the collector, it can be delayed if the network is congested. One guarantee the fast path does make is that each individual chunk of data will be received correctly or not at all. As we will see, this guarantee is enough to be useful.

On the regular "reliable path", collectors write their data in the standard Hadoop sequence file format. This format is specifically designed to facilitate parallel processing with MapReduce. To reduce the number of files and to ease analysis, Chukwa includes an "archiving" MapReduce job that groups data by cluster, date, and data type. This storage model is designed to match the typical access patterns of jobs that use the data. (For instance, it facilitates writing jobs that purge old data based on age, source, and type: "Store user logs for 14 days, and framework logs for one year.") The archiving job also detects data loss, and removes duplicate data. Repeated invocations of this job allow data to be compacted into progressively larger files over time.

This stored data can be used in a number of ways. Chukwa includes tools for searching these files. The query language allows regular-expression matches against the content or metadata of the stored data. For larger or more complex tasks, users can run customized MapReduce jobs on the collected data. Chukwa integrates cleanly with Pig, a language and execution environment for automatically producing sequences of MapReduce jobs for data analysis [18].

## 3.3 Metadata

When agents send data, they add a number of metadata fields, listed in Table 2. This metadata serves two distinct purposes: uniquely identifying a chunk for purposes of duplicate detection, and supplying context needed for analysis. Three fields identify the stream. Two are straightforward: the stream name (e.g. `/var/log/datanode`) and source host. In addition, we also tag data with the "source cluster." In both clouds and datacenters, users commonly allocate virtual clus-

ters for particular tasks and release them when the task is complete. If two different users each use a given host at different times, their logs may be effectively unrelated. The source cluster field helps resolve this ambiguity. Another field, the sequence ID, identifies the position of a given data chunk within that stream.

To these four fields, we add one more, "data type," that specifies the format of a chunk's data. Often, only a subset of the data from a given host is relevant to a given analysis. One might, for instance, only look at Hadoop Task logs. The datatype field lets a human or a program describe the logical content of chunks separately from the physical origin of the data. This avoids the need to separately maintain a table describing the semantics of each file or other physical data source.

Taken together, this metadata set allows MapReduce jobs to easily check if data is missing from a stream. (Data can be missing from a stream either for streams with reliable retransmission disabled, or as a transitory condition before a retransmission.) Missing data will show up as a gap between the sequence numbers for a pair of adjacent chunks, in precisely the same way that TCP sequence numbers allow dropped packets to be detected.

The Chukwa metadata model does not include time stamps. This was a deliberate decision. Timestamps are unsuitable for ordering chunks, since several chunks might be read from a file in immediate succession, resulting in them having identical timestamps. Nor are timestamps necessarily useful for interpreting data. A single chunk might correspond to many minutes of collected data, and as a result, a single timestamp at the chunk level would be misleading. Moreover, such timestamps are redundant, since the content of each chunk generally includes precise application-level timestamps. Standard log file formats include per-line timestamps, for instance.

## 3.4 Reliability

Fault-tolerance was a key design goal for Chukwa. Data must still arrive even if processes crash or network connectivity is interrupted. Our solution differs substantially from other systems that record logs to distributed storage and is a major contribution of this work. Rather than try to make the writer fault-tolerant, we make them stateless, and push all state to the hosts generating the data.

Handling agent crashes is straightforward. As mentioned above, agents regularly checkpoint their state. This checkpoint describes every data stream currently being monitored and how much data from that stream has been committed to the data sink. We use standard daemon-management tools to restart agents after a crash. When the agent process resumes, each active adaptor is restarted from the most recent checkpoint state. This means that agents will resend any data sent but not yet committed or committed after the last checkpoint. These duplicate chunks will be filtered out by the archiving job, mentioned above.

File tailing adaptors can easily resume from a fixed offset in the file. Adaptors that monitor ephemeral data sources, such as network sockets, can not. In these cases, the adaptor can simply resume sending data. In some cases, this lost data is unproblematic. For instance, losing one minute's system metrics prior to a crash does not render all subsequent metrics useless. In other cases, a higher reliability standard is called for. Our solution is to supply a library of "wrapper" adaptors that buffer the output from otherwise-unreliable data sources. Currently, users can choose between no buffering, buffering data in memory, or write-ahead logging on disk. Other strategies can be easily implemented.

Rather than try to build a fault tolerant collector, Chukwa agents look *through* the collectors to the underlying state of the filesystem. This filesystem state is what is used to detect and recover from failure. Recovery is handled entirely by the agent, without requiring anything at all from the failed collector. When an agent sends data to a collector, the collector responds with the name of the HDFS file in which the data will be stored and the future location of the data within the file. This is very easy to compute – since each file is only written by a single collector, the only requirement is to enqueue the data and add up lengths.

Every few minutes, each agent process polls a collector to find the length of each file to which data is being written. The length of the file is then compared with the offset at which each chunk was to be written. If the file length exceeds this value, then the data has been committed and the agent process advances its checkpoint accordingly. (Note that the length returned by the filesystem is the amount of data that has been successfully replicated.) There is nothing essential about the role of collectors in monitoring the written files. Collectors store no per-agent state. The reason to poll collectors, rather than the filesystem directly, is to reduce the load on the filesystem master and to shield agents from the details of the storage system. On error, agents resume from their last checkpoint and pick a new collector. In the event of a failure, the total volume of data retransmitted is bounded by the period between collector file rotations.

The solution is end-to-end. Authoritative copies of data can only exist in two places: the nodes where data was originally produced, and the HDFS file system where it will ultimately be stored. Collectors only hold soft state; the only "hard" state stored by Chukwa is the agent checkpoints. Figure 2 diagrams the flow of messages in this protocol.

| Field | Meaning | Source |
|---|---|---|
| Source | Host where Chunk was generated | Automatic |
| Cluster | Cluster host is associated with | Configured by user per-host |
| Datatype | Format of output | Configured by user per-stream |
| Sequence ID | Offset of Chunk in stream | Automatic |
| Name | Name of data source | Automatic |

Table 2: The Chukwa Metadata Schema

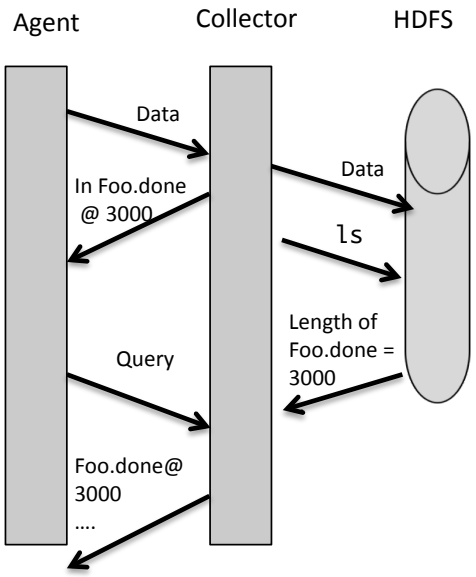


Figure 2: Flow of messages in asynchronous acknowledgement. Data written through collector without waiting for success. Separately, collectors check lengths of written files, and report this back to agents.

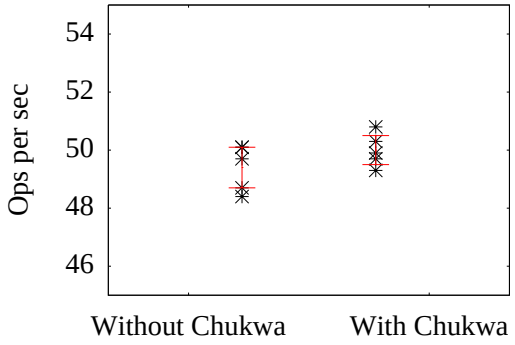


Figure 3: Cloudstone benchmark scores (HTTP requests per second), with and without Chukwa monitoring

## 4 Evaluation

In this section, we will demonstrate three properties. First, Chukwa imposes a low overhead on the system being monitored. Second, Chukwa is able to scale to large data volumes. Third, that Chukwa recovers correctly from failures. To verify these properties, we conducted a series of experiments at scale on Amazon's Elastic Compute Cloud, EC2. Using EC2 means that our hardware environment is well-documented, and that our software environment could be well controlled. All nodes used the same virtual machine image, running Ubuntu Linux, with a 2.6.21 kernel. We used version 0.20.0 of the Hadoop File System.

### 4.1 Overhead of Monitoring

To measure the overhead of Chukwa in production, we used Cloudstone, a benchmark [24], designed for comparing the performance of web application frameworks and configurations. Each run takes about ten minutes to complete and outputs a score in requests handled per second for a standardized simulated workload. The version we used starts a large number of Ruby on Rails processors, backed by a MySQL database. We used a 9-node cluster, with Chukwa running on each host. Each node was an EC2 "extra large" (server class) instance. Chukwa was configured to collect console logs and system metrics. In total, this amounted to 60 KB per minute of monitoring data per node.

Our results are displayed in Figure 3. As can be seen, the runs with and without Chukwa were virtually indistinguishable. All of the runs within Chukwa performed within 3% of the median of non-Chukwa runs. This shows that the overhead of monitoring using Chukwa is quite modest. One run each with and without Chukwa failed, due to a bug in the current Cloudstone implementation. These have been excluded from Figure 3.

To test overhead with other workloads, we ran a series of Hadoop jobs, both with and without Chukwa. We used a completely stock Hadoop configuration, without any Chukwa-specific configuration. As a result, our results reflect the experience that a typical system would have when monitored by Chukwa. We used a 20-node Hadoop cluster, and ran a series of random-writer and word-count jobs, included with the standard Hadoop distribution. These jobs are commonly used as Hadoop bench-

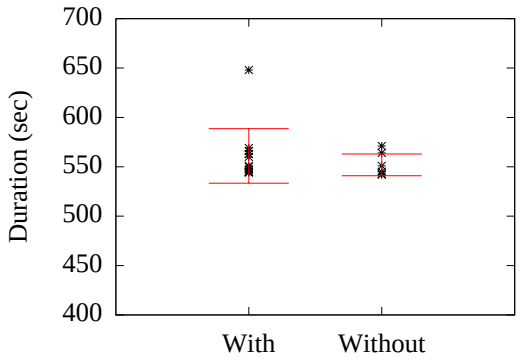


Figure 4: Hadoop job execution times, with and without Chukwa monitoring

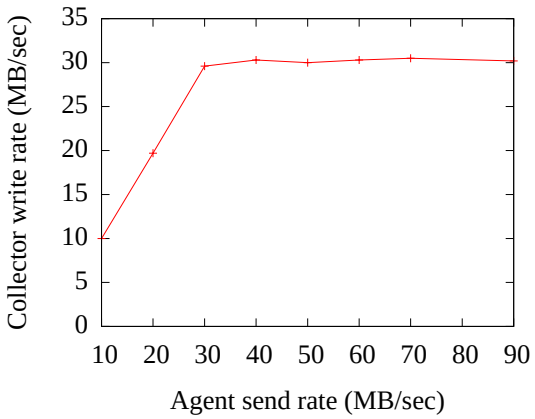


Figure 5: Throughput as a function of configured maximum send rate, showing that Chukwa can saturate the underlying filesystem. Fan-in of 200-1.

marks and their performance characteristics are well understood [32]. They first produced, then indexed, 50 GB of random text data. Each pair of jobs took roughly ten minutes to execute. Chukwa was configured to collect all Hadoop logs plus standard system metrics. This amounted to around 120 KB/min/node, and an average of 1296 adaptors per node.

Of this data, roughly two-thirds was task logs, and most of the rest was Hadoop framework logs. This is in accord with the internal Yahoo! measurements quoted in [7]. The IO performance of EC2 instances can vary by a few percent. We used the same instances throughout to control for this. The first job, run with Chukwa, was noticeably slow, presumably due to EC2 disk effects. All subsequent sequences of runs appear indistinguishable. Statistically, our results are consistent with Chukwa imposing no overhead. They effectively rule out the possibility of Chukwa imposing more than a 3% penalty on median job completion time.

## 4.2 Fan-in

Our next round of experiments was designed to verify that Chukwa collectors could handle the data rates and degree of fan-in expected operationally. Recall that our goal was to use no more than 5% of a cluster's resources for monitoring. Hence, designating 0.5% of machines as Chukwa collector and storage nodes is reasonable. This works out to a 200-to-1 fan-in.

We measured the maximum data rate that a single collector could handle with this degree of fan-in by conducting a series of trials, each using a single collector and 200 agents. In each run, the collector was configured to write data to a five-node HDFS cluster. After 20 minutes, we stopped the agents, and examined the received data.

As can be seen in Figure 5, a single collector is able to handle nearly 30 MB/sec of incoming data, at a fan-in of 200-to-1. However, as the data rate per agent rises above that point, collector throughput plateaus. The Hadoop filesystem will attempt to write one copy locally, meaning that in our experimental setup, Collector throughput is limited by the sequential-write performance of the underlying disk. From past experiments, we know that 30 MB/sec is a typical maximum write rate for HDFS instances on EC2 in our configuration. Chukwa achieves nearly the maximum possible data rates on our configuration. We checked for lost, duplicate, and corrupted chunks — none were observed.

## 4.3 Scale

Hadoop and its HDFS file system are robust, mature projects. Hadoop is routinely used on clusters with thousands of nodes at Yahoo! and elsewhere. HDFS performs well even with more than a thousand concurrent writers, e.g. in the Reduce phase of a large distributed sort. [19]. In this section, we show that Chukwa is able to take advantage of these scaling properties. To do this, we started Hadoop clusters with a range of sizes, and a Chukwa collector on each Hadoop worker node. We then started a large number of agents, enough to drive these collectors to saturation, and measured the resulting performance. The collectors and HDFS DataNodes (workers) were hosted on "medium CPU-heavy" instances. The agent processes ran on "small" instances.

Rather than collect artificial logs, we used the output from a special adaptor emitting pseudorandom data at a controlled rate. This adaptor chooses a host-specific pseudorandom seed, and stores it in each chunk. This allows convenient verification that the data received came from the expected stream and at the expected offset in the stream.

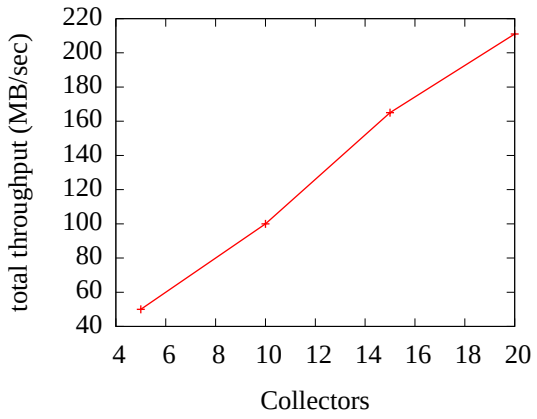


Figure 6: Aggregate cluster data collection rate, showing linear scaling.

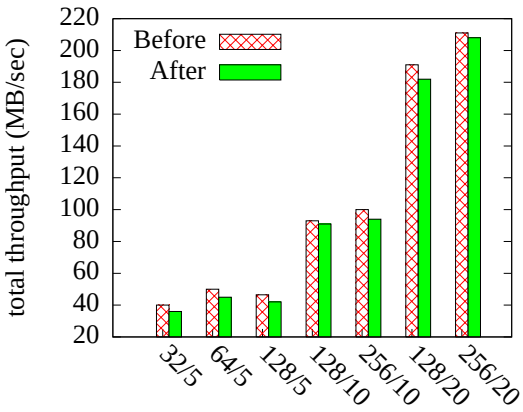


Figure 7: Performance before and after killing two collectors, showing modest degradation of throughput. Labels represent numbers of agents/collectors.

Our results are displayed in Figure 6. Aggregate write bandwidth scales linearly with the number of DataNodes, and is roughly 10 MB/sec per node — a very substantial volume of log data. This data rate is consistent with our other experiences using Hadoop on EC2. In this experiment, the Chukwa collection cluster was largely IO-bound. Hosts had quite low CPU load and spent most of their time in the `iowait` state, blocked pending disk I/O. Chukwa is saturating the filesystem, supporting our assertion above that collector processes will seldom be the bottleneck in a Chukwa deployment.

Recall that our original goal was for Chukwa to consume less than 5% of a cluster's resources. The experiments presented here demonstrate that we have met this goal. Assume that monitoring imposes a 3% slowdown on each host. That would leave 2% of the cluster's resources for dedicated collection nodes. Given a thousand-node cluster, this would mean 20 dedicated Chukwa collectors and a 50-to-1 fan-in. Given the data rates observed in [7], each collector would only be responsible for 130 KB/sec; slightly over 1% of our measured collection capacity on a 20-node HDFS cluster. We conclude that, given 5% of a cluster's resources, Chukwa is able to easily keep up with real-world datacenter logging workloads.

## 4.4 Failure Tolerance

Fault-tolerance is a key goal for Chukwa. We ran a series of experiments to demonstrate that Chukwa is able to tolerate collector failures without data loss or substantial performance penalty. The configurations in this experiment were the same as described above, with a Chukwa collector on every HDFS node.

We began by testing Chukwa's response to the permanent failure of a subset of collectors. Our procedure was as follows: After running a test cluster for 10 minutes, we killed two collectors, and then let Chukwa run for another 10 minutes. We then stopped the agents and analyzed the results. We repeated this experiment with a variety of cluster sizes. In each case, all data had been received correctly, without missing or corrupted data. Figure 7 plots performance before and after stopping the two collectors. Having fewer collectors than Datanodes degraded performance slightly, by reducing the fraction of writes that were local to the collector.

We also tested Chukwa's response to a transient failure of all collectors. This models what would happen if the underlying filesystem became unavailable, for instance if the HDFS Namenode crashed. (The HDFS Namenode is a single point of failure that sometimes crashes, resulting in the filesystem being unavailable for a period from minutes to hours.) We began our experiment with 128 agents and 10 collectors running. After five minutes, we turned off the collectors. Five minutes later, we turned them on again. We repeated this process two more times.

We plot data received over time in Figure 8. As can be seen, data transfer resumes automatically once collectors are restarted. No data was lost during the experiment. The data rate quickly jumps to 100 MB/sec, which is consistent with the maximum rates measured above for clusters of this size.

# 5 Case Studies

In this section, we discuss operational experiences using Chukwa in several contexts: web log analysis at several technology companies and real-time Cloud monitoring at Berkeley. We show that Chukwa is a natural solution for these disparate problems.

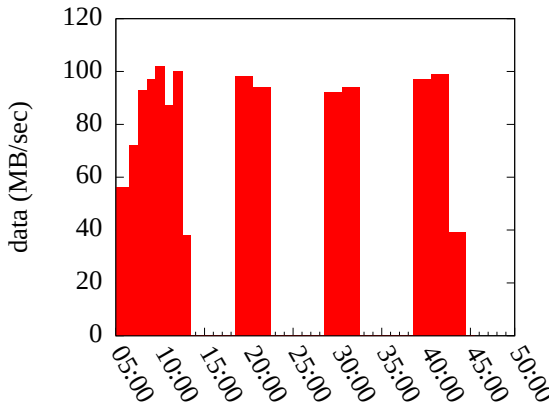


Figure 8: Data rate over time with intermittent collectors. Data transfer resumes automatically whenever collectors are available.

## 5.1 Web Log analysis

Web access logs are a particularly important class of logging data. These logs are typically line oriented, with one line per HTTP request. Automatically analyzing these logs is a core technical underpinning for web content and advertising companies. This makes analysis a good fit for Chukwa: the data volumes are large and short-turnaround automated analysis is important. We describe the experiences of two different companies: CBS Interactive and Specific Media.

CBS Interactive manages a wide range of online content, including the CBS News web site. Short-turnaround analysis allows the news room staff to monitor the popularity of stories from minute to minute, helping them gauge reader interest in particular topics. It also allows them to track the source of referrals to each story. Chukwa is a key piece of infrastructure for the underlying analysis. Content is served from a cluster of app servers, each of which writes its logs to local disk. Chukwa then copies this data into a small Hadoop cluster, where a series of Pig jobs aggregate this data and store it into a MySQL database. This database, in turn, is used by an internal web application to render data for users. Chukwa has been in use for several months and is functioning smoothly. The total volume of data is several gigabytes per day. A single collector is able to keep up with this load.

Specific Media is a leading online advertising vendor, responsible for placing advertisements on affiliate sites. Short-turnaround analytics are essential to make sure that ads are placed on sites with high likelihood of click-throughs from the advertiser's target demographic. The click-through data totals over a terabyte per day, before compression.

Chukwa is a natural fit for these uses. The data rates and past data volumes are high enough that distributed computing is necessary. Hadoop, with its easy scale-out, is a natural choice. However, blocking the production websites because of a monitoring or analysis system failure is unacceptable. As a result, the loosely-coupled Chukwa log tailing strategy is a good approach.

Both of these deployments made modifications to Chukwa to cope with site-specific needs. (These changes have been contributed back to the project.) Developers found it convenient to use the Chukwa agent process to manage parts of their logging workflows; notably CBS Interactive contributed the ability to trigger an HTTP post after every demux run in order to trigger further downstream processing.

## 5.2 Near-real-time Adaptive Provisioning

Chukwa was originally targeted at system analysis and debugging. But it can also be used for applications requiring lower latency in data delivery. One such application is adaptively provisioning distributed systems based on measured workload. SCADS, the Scalable Consistency-Adjustable Data Store, is an ongoing research project aiming to develop a low-latency data store with performance-safe queries [5]. A key component of SCADS is the "Director," a centralized controller responsible for making data placement decisions and for starting and stopping storage nodes in response to workload. Internally, SCADS uses X-Trace reports [11] as its data format. The total data volume varies from 60 to 90 KB/sec of data per node.

The SCADS development team opted to use local UDP to send the reports to Chukwa. Using TCP would have meant that SCADS might block if the Chukwa process fell behind and the kernel buffer filled up. Using the filesystem would have imposed unnecessary disk overhead. Each X-Trace report fits into a single UDP packet and in turn is sent through Chukwa as a single Chunk. This means that the Director will always see complete reports. The price for using UDP is that some kernels will discard local UDP messages under load. Some data loss is acceptable in this context, since the Director merely requires a representative sample, rather than every report.

Rather than wait for data to be visible in HDFS, the Director receives updates via fast path delivery. On boot, the Director connects to each collector, and requests copies of all reports. Once received, the reports are used to detect which hosts were involved in each read and write operation, and how long each host took to respond. Using this information, the Director is able to split up the data stored on an overloaded host, or consolidate the data stored on several idle ones. Data is typically delivered to the Director within a few seconds of being generated.

Using Chukwa in this scenario had a significant advantages over a custom-built system. While seeing data immediately is crucial to the Director, having a durable record for later analysis (potentially with MapReduce) is very helpful in tuning and debugging. Chukwa supports both, and can guarantee that all data that appeared once will eventually be stored. Using Chukwa also meant that the code for receiving data locally could be shared between this application and others.

### 5.3 Machine learning on logs

As mentioned in the introduction, one of our key goals was to enable various log analysis techniques that cannot gracefully tolerate lost data. We give an example of one such technique here. This illustrates the sort of automated log analysis Chukwa was intended to facilitate and shows why unreliable delivery of logs can poison the analysis.

Xu *et al.* have developed a machine learning approach able to detect many subtle error conditions by inspecting logs [30]. In a nutshell, their technique works as follows. Categorize the messages in a log and group them together based on whether they have a shared identifier (an ID number for an object in the system, such as a task ID.) Compare the number of messages of each type mentioning each identifier. For instance, on the Hadoop filesystem, a fixed number of "writing replica" statements should appear for each block. Seeing an unexpected or unusual number of events is a symptom of trouble.

Imperfect log statements can easily throw off the analysis. There is no easy way to differentiate a message dropped by the collection system event report from a message that was never sent because of an application bug. To conduct their experiments, Xu *et al.* copied logs to a central point at the conclusion of each experiment using `scp`. This would be unsuitable in production; logs grow continuously and the technique requires a consistent snapshot to work correctly. As a result, the largest test results reported for that work were using 200 nodes running for 48 hours. Experimental runs needed to be aborted whenever nodes failed in mid-run. There was no easy way to compensate for lost data.

Copying data off-node quickly, and storing it durably, would significantly enhance the scalability of the approach. Chukwa does precisely this, and therefore integrating this machine-learning approach with Chukwa was of practical importance. (This integration took place after the experiments described above had already been concluded.)

Adapting this job to interoperate with Chukwa was straightforward. Much of the processing in this scheme is done with a MapReduce job. We needed to add only one component to Chukwa — a custom MapReduce "input format" to hide Chukwa metadata from a MapReduce job and give the job only the contents of the collected chunks of log data. Aside from comments and boilerplate, this input format took about 30 lines of Java code. The analysis job required only a one-line change to use this modified input format.

## 6 Related Work

The Unix `syslogd` deamon, developed in the 1980s, supported cross-network logging [15]. Robustness and fault-tolerance were not design goals. The original specification for `syslogd` called for data to be sent via UDP and made no provision for reliable transmission. Today, `syslogd` still lacks support for failure recovery, for throttling its resource consumption, or for recording metadata. Messages are limited to one kilobyte, inconveniently small for structured data.

Splunk [25] is a commercial system for log collection, indexing and analysis. It relies on a centralized collection and storage architecture. It does not attempt high availability, or reliable delivery of log data. However, it does illustrate the demand in industry for sophisticated log analysis.

To satisfy this need, many large Internet companies have built sophisticated tools for large-scale monitoring and analysis. Log analysis was one of the original motivating uses of MapReduce [10]. Sawzall is a scripting language, designed for log analysis-type tasks, that simplifies writing big-data queries and that uses MapReduce as its execution engine [21]. A query language is only useful if there is data to query. While the MapReduce and Sawzall query tools have been described in the open literature, the details of log collection and management management in enterprise contexts are often shrouded in secrecy. For instance, little has been published about Google's "System Health infrastructure" tools, beyond mentioning their existence [22]. Chukwa is more comparable to these data sources, rather than to the query languages used to process collected data.

In the introduction, we mentioned a number of specialized log collection systems. Of these, Scribe is the best documented and has been used at the largest scale. Scribe is a service for forwarding and storing monitoring data. The Scribe metadata model is much simpler than that of Chukwa: messages are key-value pairs, with both key and value being arbitrary byte fields. This has the advantage of flexibility. It has the disadvantage of requiring any organization using Scribe to develop its own metadata standard, making it harder to share code between organizations.

A Scribe deployment consists of one or more Scribe servers arranged in a directed acyclic graph with a pol-

icy at each node specifying whether to forward or store incoming messages. In contrast to Chukwa, Scribe is not designed to interoperate with legacy applications. The system being monitored must send its messages to Scribe via the Thrift RPC service. This has the advantage of avoiding a local disk write in the common case where messages are delivered without error. It has the disadvantage of requiring auxiliary processes to collect data from any source that hasn't been adapted to use Scribe. Collecting log files from a non-Scribe-aware service would require using an auxiliary process to tail them. In contrast, Chukwa handles this case smoothly.

As mentioned above, Scribe makes significantly weaker delivery guarantees than Chukwa. Once data has been handed to a Scribe server, that server has responsibility for the data. Any durable buffering for later delivery is the responsibility of the server, meaning that the failure of a Scribe server can cause data loss. There can be no end-to-end delivery guarantees, since the original sender does not retain a copy. Clients can be configured to try multiple servers before giving up, but if a client cannot find a working Scribe server, data will be lost.

Another related system is Artemis, developed at Microsoft Research to help debug large Dryad clusters [9]. Artemis is designed purely for a debugging context: it processes logs *in situ* on the machines where they are produced, using DryadLINQ [31] as its processing engine. The advantage of this architecture is that it avoids redundant copying of data across the network, and enables machine resources to be reused between the system being analyzed and the analysis. The disadvantage is that queries can give the wrong answer if a node crashes or becomes temporarily unavailable. Artemis was not designed to use long-term durable storage, which requires replication off-node. Analysis on-node is also a poor fit for monitoring production services. Analyzing data where it is produced risks having data analysis jobs interfere with the system being monitored. Chukwa and Scribe, in contrast are both designed to monitor production services and were designed to decouple analysis from collection.

Chukwa is flexible enough to emulate Artemis if desired, in situations with large data volumes per node. Instead of writing across a network, agents could write to a local Hadoop filesystem process, with replication disabled. Hadoop could still be used for processing, although having only a single copy of each data item reduces the efficiency of the task scheduler [20].

Flume is another, more recent system developed for getting data into HDFS [2]. Flume was developed after Chukwa, and has many similarities: both have the same overall structure, and both do agent-side replay on error. There are some notable differences as well. In Flume, there is a central list of ongoing data flows, stored redundantly in Zookeeper. Whereas Chukwa does this end-to-end, Flume adopts a more hop-by-hop model. In Chukwa, agents on each machine are responsible for deciding what to send.

There are also a number of more specialized monitoring systems worth mentioning. Tools like Astrolabe, Pier, and Ganglia [27, 14, 17] are designed to help users query distributed system monitoring data. In each case, an agent on each machine being monitored stores a certain amount of data and participates in answering queries. They are not designed to collect and store large volumes of semi-structured log data, nor do they support a general-purpose programming model. Instead, a particular data aggregation strategy is built into the system.. This helps achieve scalability, at the cost of a certain amount of generality. In contrast, Chukwa separates the analysis from the collection, so that each part of a deployment can be scaled out independently.

## 7 Conclusions

There is widespread interest in using Hadoop to store and process log files, as witnessed by the fact that several systems have been built to do this. Chukwa improves on these systems in several ways. Rather than having each part of the monitoring system be responsible for resuming correctly after a failure, we have demonstrated an end-to-end approach, minimizing the amount of state that needs to be stored in the monitoring system. In recovering from failures, Chukwa takes advantage of local copies of log files, on the machines where they are generated. This effectively pushes the responsibility for maintaining data out of the monitoring system, and into the local filesystem on each machine. This file-centered approach also aids integration with legacy systems. Chukwa also offers the flexibility to support other data sources, such as `syslog` or local IPC.

Chukwa is efficient and practical. It was designed to be suitable for production environments, with particular attention to the cloud. Chukwa has been used successfully in a range of operational scenarios. It can scale to large data volumes and imposes only a small overhead on the system being monitored.

We have shown that Chukwa scales linearly up to 200 MB/sec. If sufficient hardware were available, Chukwa could almost certainly match or exceed the highest reported cluster-wide logging rate in the literature, 277 MB/sec. [9]. While few of today's clusters produce remotely this much data, we expect that the volume of collected monitoring data will rise over time. A major theme in computer science research for the last decade has been the pursuit of ever-larger data sets and of analysis techniques to exploit them effectively [13]. We expect this to hold true for system monitoring: given a scalable log col-

lection infrastructure, researchers will find more things worth logging, and better ways of using those logs. For instance, we expect tracing tools like XTrace and DTrace to become more common [11, 8]. Chukwa shows how to build the necessary infrastructure to achieve this at large scale.

## Availability

Chukwa is a subproject of Hadoop and is overseen by the Apache Software Foundation. All code is available under permissive license terms. At present, the Chukwa website is `http://hadoop.apache.org/chukwa`; releases can be obtained from there.

## Acknowledgments

We thank Bill Graham and Gerrit van Vuuren for describing how Chukwa is being used at CBS Interactive and Specific Media, respectively. This paper has focused on the data collection and transport side of Chukwa. Eric Yang and Jerome Boulon were responsible for the processing and display components, which are also important aspects of the overall system. The students and faculty of the RAD Lab at UC Berkeley supplied a great deal of feedback and advice.

This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

## References

[1] Scribe. http://sourceforge.net/ projects/ scribeserver/, 2008.

[2] Cloudera's flume. `http://github.com/cloudera/flume`, June 2010.

[3] M. Aharon, G. Barash, I. Cohen, and E. Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, Bled, Slovenia, September 2009.

[4] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report 2009-28, UC Berkeley, 2009.

[5] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-Independent Storage for Social Computing Applications. In *Fourth Conference on Innovative Data Systems Research*, Asilomar, CA, January 2009.

[6] D. Borthakur. HDFS Architecture. http://hadoop. apache. org/common/ docs/r0.20.0/ hdfs_design.html, April 2009.

[7] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa, a large-scale monitoring system. In *First Workshop on Cloud Computing and its Applications (CCA '08)*, Chicago, IL, 2008.

[8] B. Cantrill. Hidden in Plain Sight. *ACM Queue*, 4(1), 2006.

[9] G. F. Creţu-Ciocârlie, M. Budiu, and M. Goldszmidt. Hunting for problems with Artemis. In *First USENIX Workshop on Analysis of System Logs (WASL '08)*, San Diego, CA, December 2008.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, Volume 51(Issue 1):107–113, 2008.

[11] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. XTrace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, April 2007.

[12] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *19th Symposium on Operating Systems Principles (SOSP)*, 2003.

[13] A. Halevy, P. Norvig, and F. Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24:8–12, 2009.

[14] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, pages 321–332, 2003.

[15] C. Lonvick. RFC 3164: The BSD syslog Protocol. http://www.ietf.org/rfc/rfc3164.txt, August 2001.

[16] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[17] M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110. ACM New York, NY, USA, 2008.

[19] O. O'Malley and A. C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. http://sortbenchmark.org/ Yahoo2009.pdf, April 2009.

[20] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, Providence, RI, 2009.

[21] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming Journal*, Volume 13(Number 4/2005):277–298, 2003.

[22] E. Pinheiro, W. Weber, and L. Barroso. Failure Trends in a Large Disk Drive Population. In *5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose, CA, 2007.

[23] Rapleaf, inc. The collector. http://blog.rapleaf .com/dev/?p=34, October 2008.

[24] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Cloud Computing and Applications*, 2008.

[25] Splunk Inc. IT Search for Log Management, Operations, Security and Compliance. http://www.splunk.com/, 2009.

[26] J. Stearley. Towards informatic analysis of syslogs. *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004.

[27] R. Van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM TOCS*, 21(2):164–206, 2003.

[28] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *18th Symposium on Operating Systems Principles (SOSP)*, 2001.

[29] T. White. *Hadoop: The Definitive Guide*, pages 439–447. O'Reilly, Sebastopol, CA, 2009.

[30] W. Xu, L. Huang, M. Jordan, D. Patterson, and A. Fox. Detecting Large-Scale System Problems by Mining Console Logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[32] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.