Chukwa: A system for reliable large-scale log collection

Ariel Rabkin
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-25
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-25.html

March 5, 2010

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 companies have developed log collection systems that write to HDFS. These systems have a number of common weaknesses, induced by the semantics of the filesystem. They impose a delay, often several minutes, before data is available for processing. They are difficult to integrate with existing applications. They cannot reliably handle concurrent failures. We present a system, called Chukwa, that adds the needed semantics for log collection and analysis. Chukwa uses an end-to-end delivery model that leverages local on-disk log files when possible, easing integration with legacy systems. Chukwa 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.

1 Introduction

MapReduce [8] is a popular framework for large-scale distributed processing. One of its major principles 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) [10], or its open-source counterpart, the Hadoop Distributed File System (HDFS).

One important application of MapReduce is processing log files of various types [8, 29]. GFS and HDFS are user-level filesystems that do not implement POSIX semantics and that do not integrate with the OS filesystem layer. 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.

At least outside Google, the latter approach appears more popular. It is more flexible and has the advantage of confining the log storage-related code to a smaller portion of the overall system deployment. Several companies, such as Rapleaf, Rackspace, and Facebook [22, 29, 1], have built specialized log collection systems that take this approach to storing logs in HDFS. The system developed at Facebook, known as Scribe, is open source and therefore particularly well-known. The existence of these separate implementations is evidence that log collection is a significant problem that currently lacks a definitive solution.

1.1 The Problem

HDFS has been primarily designed to store large data sets in a modest number of large files, each of which is written once and then closed. This is a good fit for the input and output from MapReduce jobs, but is a less good fit for logs. Log data is typically spread over a large number of data sources, of varying sizes, updated sporadically. The requirement to harmonize these two sets of semantics significantly constrains the design of a log collection system. As a result, the collection systems mentioned above all share a common architecture. Unlike GFS, HDFS does not allow concurrent appends, meaning that a user process, often called a collector, fills the role. This process receives log messages from each log source and consolidates the received messages into a single file.

There are three problems with this approach to log collection. Two of them stem from tradeoffs in the design of the underlying filesystem. To get good performance, HDFS caches data aggressively and does not offer clients an `fsync`-like primitive to force collected data to the filesystem. Written data only becomes visible once a complete block (64 MB) has been written or the file has

been closed. Since the file system does not perform well with large numbers of small files, each of these collection systems keeps their output files open for a lengthy period, generally measured in minutes. This avoids the small-files problem, but imposes a delay before data becomes visible. This delay can be more than a minute in production contexts [29].

This caching also poses a reliability problem: partial blocks are not persistent. This means that special measures need to be taken to avoid losing data if a collector crashes. In some contexts, this risk is acceptable. Scribe simply risks the data loss, while warning that "some multiple component failure cases" can lead to data loss. A more robust approach (taken by [22]) is to keep a write-ahead log at the collector. This approach can still entail data loss if the collector failure is permanent.

Last, this collector-based architecture is hard to integrate with legacy applications. To avoid complexity at the collector, each collection system has a fixed and narrow API. For instance, Scribe's data model is that messages are a key-value pair sent via Thrift RPC. Writing and configuring wrappers for every existing data source is likely to be both troublesome and error-prone. Having a network API as the primary interface to the log collection system means a great deal of configuration, for instance, primary and backup server addresses, spread over every service being monitored.

Adding a "record append" operation to the filesystem would go a long way towards solving the first two of these problems. This feature appears to be quite difficult to get right, however. The simpler case of single-writer appends in HDFS has been in development for nearly two years and is still not available in current versions of Hadoop, despite the efforts of large teams of sophisticated developers at Yahoo! and Facebook [13]. It is therefore highly desirable to find a simpler approach to reliable logging.

1.2 Our solution

Our system, Chukwa, was originally developed in response to the need for a distributed log collection system at Yahoo's Hadoop development and operations group. It is an open source project that our group at UC Berkeley has adopted and substantially extended. In a previous publication, we described our initial goals and our prototype implementation. In this paper, we describe Chukwa's reliability mechanisms, present detailed measurements, and describe real-world experiences.

While logging to HDFS was our original motivation, both the problem and solution are more broadly applicable. Several other distributed storage systems (including GFS [10]) were built to support latency-insensitive batch workloads and and more such systems may be developed over time. HDFS is unlikely to be the last storage system ever developed that has long-latency writes and that is tuned for large files. Chukwa shows that it is possible to support logging workloads on these systems, even without sophisticated APIs, by adding a specialized collection system on top of the filesystem. We require neither a record append operation nor an `fsync`-like operation, both of which have proven very hard to implement at scale.

Chukwa addresses all three problems described above that hamper current log collection systems. We introduce a "fast path" delivery model to make data available more promptly, bypassing the filesystem for these clients. We achieve reliability end-to-end, using existing local copies of logs whenever possible. We introduce an agent process on each host being monitored that can collect log data from legacy systems in standardized way, pulling complexity into the monitoring system and away from the users. While each of these improvements to log collection could be made separately, they are mutually reinforcing. Emphasizing the collection of existing on-disk log files aids both reliability and support for existing applications. The data model that we use to achieve reliability also makes our fast path delivery much more useful.

In many environments, applications are configured to write their logs to the local filesystem. This observation was based on our discussions with developers at both UC Berkeley and at Yahoo!, inc. In both environments, developers wanted to have copies of their log files on the machines where those files are produced. Local files are very simple to understand and reason about. They are easy to produce, for instance by redirecting the standard out or standard error streams from processes without reconfigurable logging. There is often a substantial degree of tool support for collecting or analyzing these files. For instance, Hadoop makes local log files visible via an HTTP interface on each machine. Perhaps most importantly, local logging is very robust. If a network service is inaccessible, then the logging application is responsible for buffering or retrying. This is potentially a significant complexity to insert into every application generating logs. In contrast, filesystem writes will always succeed if space is available. Putting logs on a separate partition makes it easy to reserve sufficient space for them.

A key insight behind Chukwa is that the monitoring system can use this on-disk copy for reliability. Having a durable local copy lets the collection system use an end-to-end failure recovery model. From the point of view of the client system containing the logs, data is written to HDFS asynchronously. Clients periodically check whether their logs were copied to HDFS correctly. If not, the local copy is available for re-trying. Turning

this insight into an effective monitoring system required us to handle two challenges: The bookkeeping necessary for recovery must be efficient, even at large scale and the mechanism must be adaptable to cases where the data to be collected is not initially stored in log files. As we will show in this paper, Chukwa meets these challenges.

This work is timely for two reasons. The development of automated log analysis (such as [30, 2, 15] 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 [3], 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. Section 3 describes our concrete implementation and Section 4 presents some quantitative measurements. Section 5 describes our experiences integrating Chukwa with applications; Section 6 discusses how we use the collected data. We then describe related work in Section 7, and conclude in Section 8.

2 Design Goals

We now lay out our goals and assumptions in more detail. These 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` [14].
- 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.
- No matter how intense a bust of log writes, the resource consumption of the monitoring system remain with its resource bounds.
- 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 [5].

2.2 Delivery Models

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.

Some log analysis jobs are very sensitive to missing data. In general, whenever the absence of a log message is significant to an analysis, then 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. To support these sorts of applications, we made reliable delivery a core goal for Chukwa Our reliability goal is that data will be correctly written to the distributed filesystem even in the presence of arbitrary combinations of transitory failures.

Not all sources 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. Chukwa's reliability model is designed to cope gracefully with ephemeral data sources, whose output need not be persistent across a crash. Administrators can select between different reliability models, at per-data-source granularity.

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. 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

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

these goals. Like the other systems of this type, we introduce auxiliary processes between the log data and the 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 [14, 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 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 backpressure 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. he 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.

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.

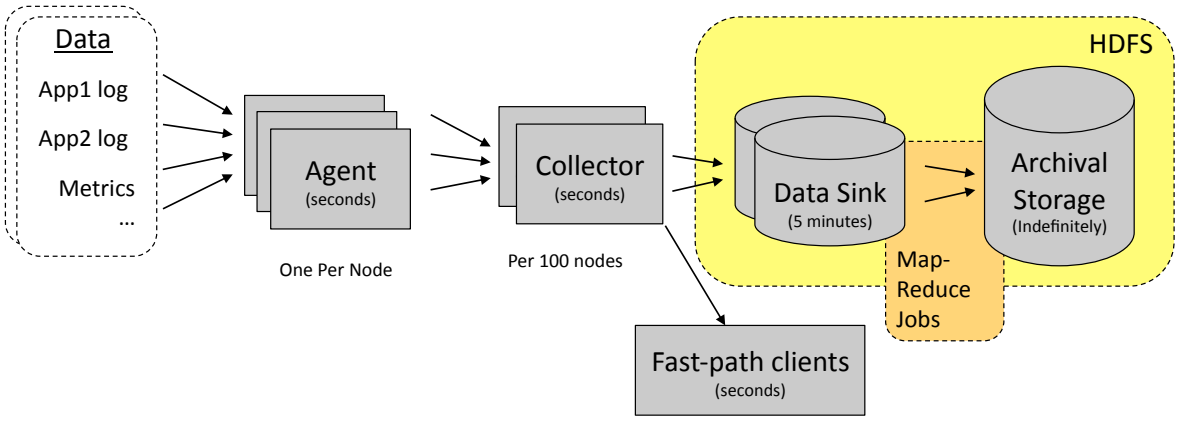

Figure 1: The flow of data through Chukwa, showing retention times at each stage.

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 even very large datacenter services.

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 [17].

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 clusters 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 HadoopTask 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.

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

Field	Meaning	Source
Source	Hostname 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

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.

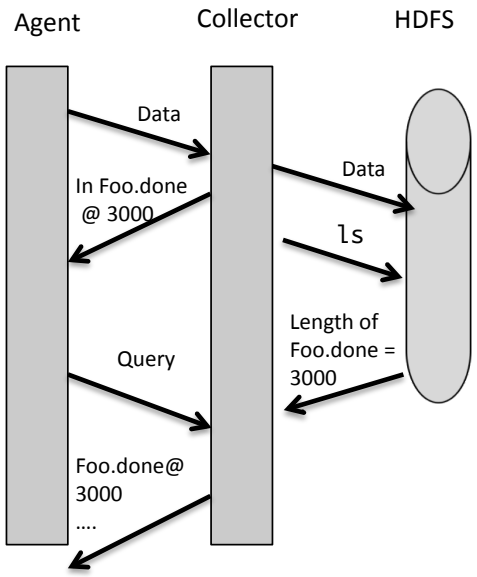

Figure 2: Flow of messages in asynchronous acknowledgement. Data written through collector without waiting for success. Separately, collectors check lengths of written files, and report this back to agents.

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, the most recently released version at the time.

4.1 Overhead of Monitoring

To measure the overhead of Chukwa in production, we used Cloudstone, a benchmark [23], 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 indis-

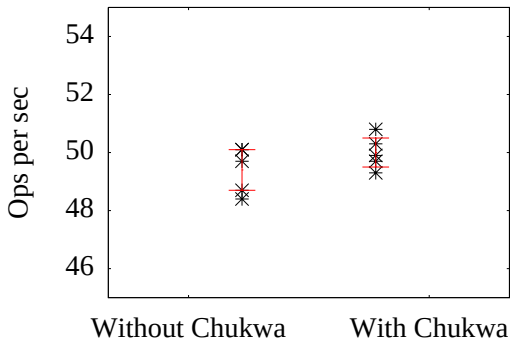

Figure 3: Cloudstone benchmark scores (in HTTP requests per second), with and without Chukwa

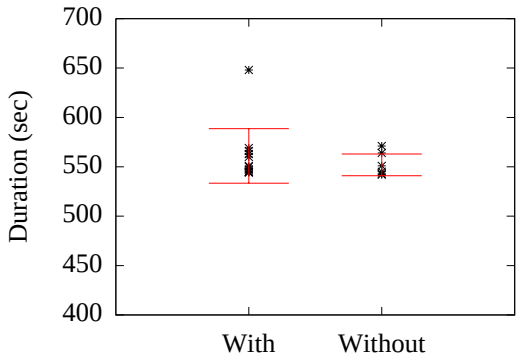

Figure 4: Hadoop job execution times, with and without Chukwa

tinguishable. 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 benchmarks 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 2 KB/sec/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 close accordance with the internal Yahoo! measurements quoted in [5]. The IO performance of EC2 instances can vary by a few percent. We used the same instances throughout to control for this. Another quirk of EC2 is that nodes are slow to "warm up" – the first disk write to a block will be much slower than subsequent writes. To control for this effect, we tested first with Chukwa, then without, then with again. In each test, we ran six pairs of random-write and word-count. The results are plotted in Figure 4, with standard-deviation bars around the mean. 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 rule out the possibility of Chukwa imposing more than a 3% penalty on median job completion time.

We verified this observation statistically. Ignoring the first anomalous run, only 2 of 11 of the runs with Chukwa took more than 3% longer than the median for runs without Chukwa. Chukwa imposed a 3% or greater penalty, the chances of seeing so few chances of observing this would be slightly over 0.03. (We assume successive runs are independent, after the initial warmup.) We conclude that Chukwa, in our configuration, very likely imposes less than a 3% overhead.

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

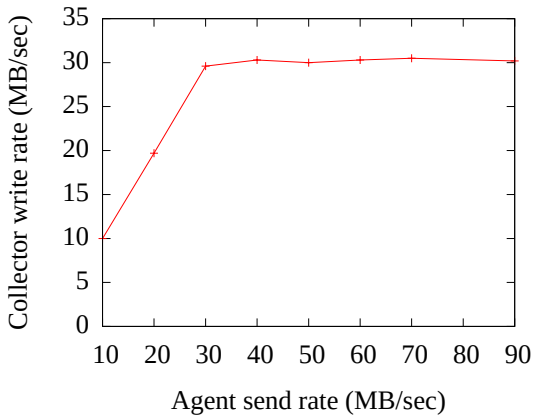

Figure 5: Throughput as a function of configured maximum send rate, showing that Chukwa can saturate the underlying filesystem. Fan-in of 200-1.

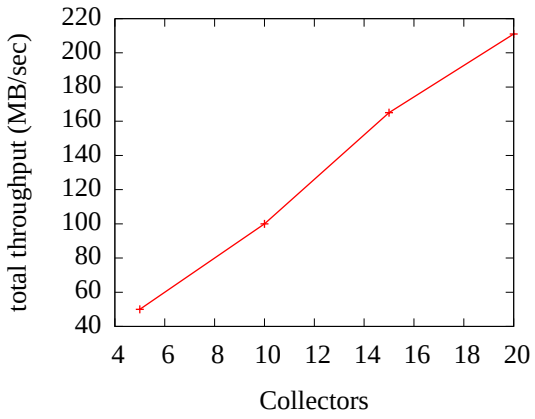

Figure 6: Aggregate cluster data collection rate, showing linear scaling.

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. [18].

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.

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 [5], 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. Fig-

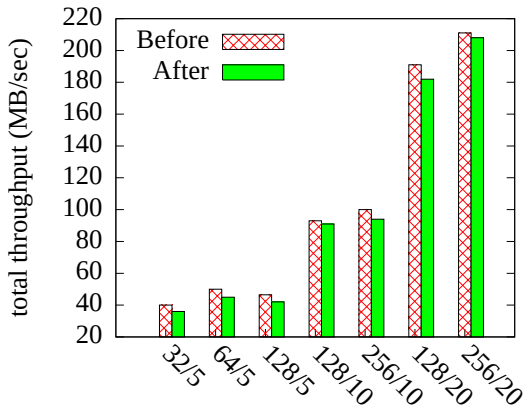

Figure 7: Performance before and after killing two collectors, showing modest degradation of throughput. Labels represent numbers of agents/collectors.

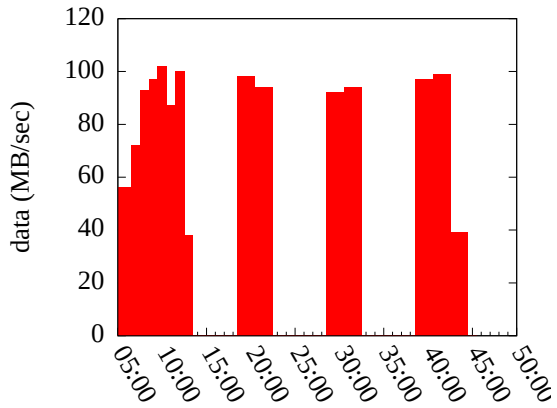

Figure 8: Data rate over time with intermittent collectors. Data transfer resumes automatically whenever collectors are available.

ure 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 Application Integration

In the previous section, we described some experimental evidence that Chukwa has low overhead, scales to large data volumes, and recovers correctly from errors. In this section, we describe our operational experiences using Chukwa to monitor various applications. These case studies will demonstrate that Chukwa can be easily adapted to a variety of collection scenarios. In the next section, we will show that it can support a wide variety of analysis and data processing techniques.

When Chukwa is used to harvest logs from the filesystem, it needs to be told which files to monitor somehow. Doing so is the chief challenge in integrating an application with Chukwa. We have used Chukwa to monitor a range of different applications, both in the cloud and on statically-configured hosts. There are two broad approaches to configuring Chukwa: statically and dynamically. We give two examples of the former, and one of the latter. Taken together, these examples will show that Chukwa is easy to integrate with a wide range of deployment scenarios.

5.1 Static configuration

Static configuration works well when the log files to be watched or the metrics to be collected are known in advance. It works especially well in the cloud. On EC2 and similar environments, all services run in virtual machines. The configuration of these VM images is often centralized in a configuration management system. In our environment, the configurations for Chukwa and the system being monitored are typically stored in the same file. This reduces the opportunities for an administrator to inadvertently misconfigure Chukwa.

Cloudstone, as mentioned above, is a benchmark developed for comparing the performance of web application frameworks and configurations. This involves a number of separate pieces: web servers, application servers, databases, workload generators, and so forth. Each of these generates log files, that have been very useful in tracking down difficulties with the benchmark. The development team uses Chukwa to collect these files. Since each file has a fixed name, the simplest approach to take was to statically configure Chukwa with the name of each file to monitor.

SCADS, the Scalable Consistency-Adjustable Data Store, is an ongoing research project aiming to develop a

low-latency data store with performance-safe queries [4]. 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 [9] 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. Configuration was straightforward. Each agent was statically configured to run an adaptor that listens for UDP messages on the appropriate port. Other adaptors, running alongside, collect system metrics so that the performance of the system can be interpreted.

5.2 Dynamic Configuration

Sometimes, however, the names of the files to be monitored cannot be determined in advance. This is the case for Hadoop. In addition to the Hadoop framework logs, each MapReduce task generates its own log files. These files are created when each task starts and are active only for the lifetime of the task, often less than a minute. Their names and locations cannot be determined in advance, since they include the task ID, which is only selected at run-time.

Continuing to tail the log from a completed task is wasteful, since that file will not be written to after the task completes. Instead, we opted to have Hadoop dynamically control the Chukwa agent, starting and stopping adaptors as needed. Fortunately, Hadoop includes "hooks", allowing us to insert code that will be run when tasks start and stop. This code is responsible for starting and stopping file collection as tasks begin and end. All communication with Chukwa is handled asynchronously by a background thread to avoid blocking execution of Hadoop. If Chukwa is down, this thread is responsible for retrying.

6 Using the Data

Chukwa supports a variety of uses for data. We discuss four of these uses here: log warehousing, anomaly detection, graphical display, and adaptive provisioning of distributed systems.

6.1 Warehousing

Perhaps the simplest use of a log collection system is to simply archive logs for later *ad-hoc* inspection. Particularly in the cloud, leaving logs scattered across a large number of (virtual) hosts is very inconvenient. In clouds, hosts are often only used for a short time and then deallocated [3]. If logs are not copied off-node before the host is deallocated, they will be gone permanently. Hence, centralized archiving is crucial.

Our lab uses Chukwa for this purpose. Our standard virtual machine image has Chukwa support included. A single configuration setting will install and configure Chukwa agents at boot. System metrics are collected automatically, as are any user-designated logs. The "cluster" field in the Chukwa metadata is specified by the user when the cluster is allocated. The archiver job then groups received data using this field. This means that results from different users or experimental runs can be easily distinguished. Agents upload their data to collectors backed by a local Hadoop cluster at our institution.

The system has been very dependable. We have had no enforced downtime despite using the system to transfer gigabytes per day on average. There have been no reported complaints about interference with other uses of the departmental cluster, nor has the presence of regular CPU-heavy machine learning workloads interfered visibly with log collection and storage. Our users report broad satisfaction. For November, the most recent month before students left for vacation, the average volume of data collected was half a gigabyte per day.

The Chukwa architecture is a particularly good fit for monitoring experiments in the cloud. The agent/collector/processing architecture naturally takes into account the split between the system-being-analyzed and the analysis system. The system being monitored, which can span several hundred nodes, can be deallocated at the end of the experimental run while the monitoring data is stored durably on a less-elastic cluster. This way, data can be analyzed over time, even after the cluster where the logs were generated has been deallocated.

6.2 Machine Learning

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.

One powerful log analysis technique is to categorize and count the number of events associated with a particular object in a system. 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. This analysis technique can be automated: a combination of static analysis and machine learning is able to pick out the patterns that indicate true errors from the normal variation of correct behavior. This technique has been shown to work effectively in spotting system problems at large scale [30].

But missing or duplicate log statements can easily throw off the process. The analysis is unable to differentiate a missing event report from an event that should have happened, but did not. 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.

In that work, 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.

6.3 Graphical Display

An important motivating use for Chukwa was supplying system performance data for visualization. One tool that does this is HICC, the Hadoop Infrastructure Care Center. HICC is a flexible, web-based "portal-style" interface for visualizing system performance data. HICC was developed as a Chukwa subcomponent, although it can be (and has been) used independently. HICC can display traditional system metrics, such as free memory, cpu load, disk writes per second as well as application-layer statistics like the number of rack-local Map tasks,

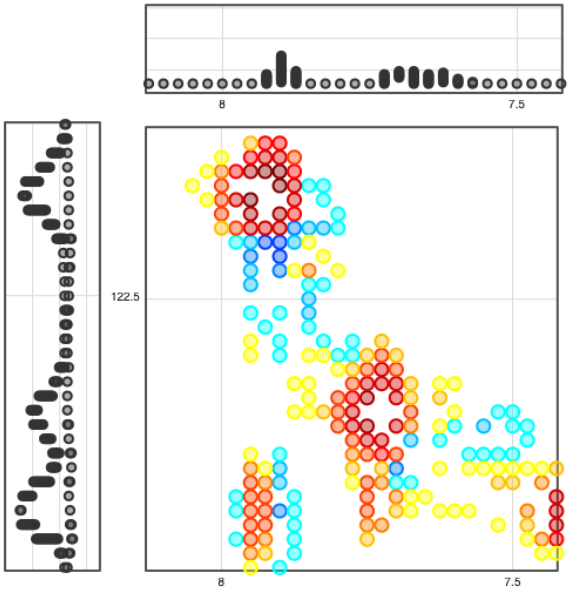

Figure 9: HICC using Mochi to display heatmap: a visualization that required Chukwa.

the number of Hadoop block moves, and so forth. While existing monitoring systems could be used to collect the former set of metrics, the latter can only be computed by inspecting logs, which requires a tool like Chukwa.

HICC can do more than just graph time-series data. HICC also includes the SALSA state-machine extraction tool and the Mochi visualization framework, developed at CMU [25, 26]. SALSA builds a state-machine model of job execution using information the console logs from each node. Mochi then displays the results in various ways. The "heatmap" visualization (Figure 9) shows which pairs of nodes are exchanging data blocks. This can only be gleaned from the logs, since it is not tracked centrally. This visualization provides insight into whether or not a job is evenly spreading network traffic across the cluster.

HDFS has high latency for read requests, and therefore performs sluggishly on interactive query workloads. Instead of serving data from HDFS directly, HICC pulls data from an SQL database. This database is populated using batch inserts prepared by MapReduce jobs run against the data collected with Chukwa. This MapReduce job runs every five minutes by default. As a result, displayed data will be at least five minutes behind real-time. HICC was designed to support applications like cluster performance tuning and Hadoop job execution visualization, where this delay is not problematic.

HICC does not currently require Chukwa's reliable delivery guarantees. It does, however, rely on Chukwa to collect data and make it available for MapReduce processing. HICC demonstrates that this architecture

works effectively to facilitate rapid-turnaround MapReduce processing of logs.

6.4 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. Previously, we described configuring Chukwa to monitor SCADS, the experimental data store. Here, we discuss how the data is used. This example demonstrates the utility of Chukwa's "fast path" delivery model.

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.

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.

7 Related Work

The Unix `syslogd` deamon, developed in the 1980s, supported cross-network logging [14]. 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 [24] 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, and the associated Sawzall scripting language [8, 20]. While these frameworks have been described in the open literature, the details of log 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 [21].

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. It is also the only one that is open source, and hence the only one we can inspect in detail. 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 policy 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.

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 [7]. 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 [19].

There are also a number of more specialized monitoring systems worth mentioning. Tools like Astrolabe, Pier, and Ganglia [27, 12, 16] 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.

8 Conclusion

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 present 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. Unlike other log collection systems, Chukwa facilitates near-real-time processing as well as reliable storage in a distributed file system.

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.

Chukwa is a part of the Hadoop software ecosystem, and relies on several other Hadoop subprojects. We assume the use of HDFS as a storage layer, MapReduce as a computation model, and Pig as a high-level data processing language. Each of these systems is stable and has a large user community. These systems will continue to improve and Chukwa will reap the benefits.

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. [7]. 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 [11]. We expect this to hold true for system monitoring: given a scalable log collection 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 [9, 6]. Chukwa provides the necessary infrastructure to achieve this at large scale.

Acknowledgments

The students and faculty of the RAD Lab at UC Berkeley supplied a great deal of feedback and advice. We particularly thank Armando Fox, Matei Zaharia, Archana Ganapathi, and Kristal Curtis. Eric Yang and Jerome Boulon were the two other major contributors to Chukwa. HICC, described in Section 6.3, is largely their work.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

[13] H. Kuang. Revisit appends. https://issues.apache.org/ jira/browse/ HDFS-265, April 2009.

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

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: Analyzing Logs as StAte Machines. In *First USENIX Workshop on Analysis of System Logs (WASL '08)*, San Diego, CA, December 2008.

[26] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop. In *Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, San Diego, CA, June 2009.

[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.