# Project Hoover: Auto-Scaling Streaming Map-Reduce Applications

Rajalakshmi Ramesh
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
rrajalakshmi@gatech.edu

Liting Hu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
foxting@gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
schwan@cc.gatech.edu

## ABSTRACT
Real-time data processing frameworks like S4 and Flume have become scalable and reliable solutions for acquiring, moving, and processing voluminous amounts of data continuously produced by large numbers of online sources. Yet these frameworks lack the elasticity to horizontally scale-up or scale-down their based on current rates of input events and desired event processing latencies. The Project Hoover middleware provides distributed methods for measuring, aggregating, and analyzing the performance of distributed Flume components, thereby enabling online configuration changes to meet varying processing demands. Experimental evaluations with a sample Flume data processing code show Hoover's approach to be capable of dynamically and continuously monitoring Flume performance, demonstrating that such data can be used to right-size the number of Flume collectors according to different log production rates.

## Categories and Subject Descriptors
D.4.1 [**Operating Systems**]: Process Management – *scheduling*; D.4.7 [**Operating Systems**]: Organization and Design – *distributed systems*.

## General Terms
Management, Performance, Design, Experimentation.

## Keywords
Flume OG, Pastry, Scribe, Queuing Model.

## 1. INTRODUCTION
Web 2.0 companies like Facebook, Yahoo, LinkedIn, and Twitter generate large amounts of log data, including (1) user activity events like clicks, comments, or sharing, (2) operational metrics like call latency and errors, and (3) system metrics like CPU, memory, and network utilization. Such data is invaluable for debugging, performance management, and for commercial reasons. Consider for instance, an e-commerce website that collects logs to monitor the number of users who are currently viewing a particular product. Using this data, the company can increase sales by running micro-promotions that offer, say "20% off", if more than 5000 users are currently viewing the same product.

Recent years have seen the development of distributed log aggregators specialized for collecting and processing online log data, such as Facebook's Scribe [3], Yahoo's Chukwa [10], and Cloudera's Flume [2]. These systems convert log entries into events, which are then aggregated and processed by distributed sets of agents in multi-tier frameworks backed by key-value stores like HBase [5] and distributed file systems like HDFS [6].

Online log aggregators face challenges. First, their processing capabilities should be horizontally scalable -- up or down -- based on current volumes of input logs. Such elasticity is important because events will be delayed or even lost if the aggregate consumption rate of intermediate processing nodes, called `collectors' in Flume, cannot keep pace with the rate at which log events are produced. Second, there is a need for load balancing across different sets of collectors, when log input rates are not evenly distributed across the system's many sources. Third, such elasticity must function at volumes up to the few billion messages a day, as companies like Facebook collect everything from access logs, to performance statistics, to actions going to its News Feed. At these scales, however, the online monitoring required for elasticity at this scale is challenging, particularly for current commercial approaches that expose metrics through JMX MBeans [8], where each MBean's attributes and operations are externally accessed through RMI [9]. RMI does not perform well at large scale due to the overheads of its registration logic, serialization, and its slow failure detection and cleanup. Recent solutions like Jolokia [7] address this by exposing MBeans over HTTP, Cloudera's Flume allows users to gather node metrics via HTTP or by injecting them as a separate data flow along with the data being processed. However, those additional data flows add unmanaged overhead to the streaming data processing subsystem. Finally, research approaches like those described in [15] constitute potential solutions, but have not yet been deployed in commercial settings.

To enable auto-scaling online web log processing systems, we present Project Hoover, which is middleware that addresses the above monitoring challenges by integrating the Pastry/Scribe multicast framework [14] [11] with Cloudera's Flume log processing system.

*Novel approach to online metrics gathering*. Rather than using an additional internal data flow to gather metrics, Hoover creates a separable external channel to export metrics to an `aggregator' that operates alongside Flume's central management master used for dynamic reconfiguration of Flume components. This is implemented via a light-weight group communication and event notification system (Scribe) built on a peer-to-peer overlay (Pastry).

*Online assessment of Flume component `health' enables auto-scaling*. The aggregator uses dynamically collected metrics to assess operational characteristics of the Flume application's execution. Specifically, this paper demonstrates the use of such metric data to model Flume components as a network of queues and measuring variables like average input rates, output rates, queue lengths, etc. This information is then used to auto-scale Flume to match its aggregate processing capacity to dynamically changing log input rates, to maintain desired end-to-end processing latencies.

Performance measurements demonstrate the efficiency of Hoover's online monitoring and analysis, as well as its utility for

adjusting Flume performance to current conditions and needs. Specifically, they show only a small increase in anycast round-trip times as the number of nodes in the Pastry-based monitoring overlay increases. Further, because the average aggregation time per node is only on the order of tens of milliseconds, it is possible to aggregate statistics from a large number of nodes within time intervals of only a few minutes. Consequently enabled scale-up/down is shown useful via a simple auto-scaler implemented based on a Secant root finding method. It predicts the number of collectors required to maintain the health of the Flume system, by providing a good approximation for Flume configurations with high average queue lengths and low health scores.

The remainder of this paper is organized as follows. Section 2 discusses background and related work. Section 3 describes Hoover‘s design. Section 4 evaluates Hoover with experiments. We conclude with directions for future work in Section 5.

# 2. BACKGROUND AND RELATED WORK

We first explain the design pattern for today's multi-tier log processing systems like Flume. We then discuss related literature.

## 2.1 BACKGROUND

Flume is a distributed service for collecting and processing large amounts of log data generated by clients, ultimately placed into some persistent store for later use. Figure 1 shows a typical deployment of Flume comprised of three tiers: (1) the agent tier generates events from client logs; (2) collectors aggregate events from separate data logs and forward them to (3) the storage tier comprised of HBASE and the Hadoop Distributed File System (HDFS).

Every node in Flume has a source and a sink. The source tells it where to collect data, while the sink tells it where to send the data. A separate process, called the Flume master, is the central management point; it directs data flows by assigning source/sink configurations to all nodes, and it communicates dynamic configuration updates. Sinks can additionally be configured with `decorators' that perform simple processing on data. For example, network throughput can be increased by batching events and then compressing them before moving them to the sink.

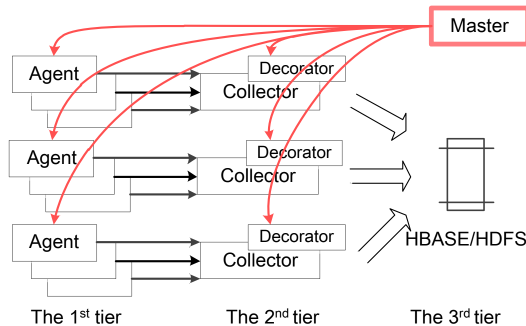


**Figure 1. Architecture of the Flume System.**

## 2.2 RELATED WORK

Facebook's Scribe [3] and Yahoo's Chukwa [10] gather logs based on the "push" model. Scribe uses local servers running on each node, in order to aggregate online logs and send them to a central collector (or to multiple collectors). Yahoo's Chukwa is built on top of the Hadoop distributed file system and the MapReduce framework. It uses a push model in which each frontend node sends logs to a set of collectors over sockets. The collectors write log entries to HDFS. LinkedIn's Kafka [13] gathers logs based on a "pull" model. A stream of messages of a particular type is defined by a topic, and a producer can publish messages to a topic. The published messages are then stored at a set of servers called brokers, which periodically write data into HDFS.

The systems outlined above facilitate reliable, scalable, efficient, and time critical aggregation and storage of live data. However, we are not aware of their ability to auto-scale their numbers of collectors based on log volume changes or workload imbalance. This will result in variable latencies in moving log data and raises the possibility of data loss and failure of collectors when they are overwhelmed.

Existing commercial log monitoring systems expose a limited set of metrics through JMX MBeans [8], where metric collection typically involves injecting JMX metrics into systems like Ganglia [4], Amazon CloudWatch [1], etc. Ganglia has been used at large scale to collect summary operational statistics in grids and clusters. However, its membership management uses native IP Multicast to communicate with its peer nodes, which is not appropriate for scale-out datacenter systems in which components join and leave frequently. Preferable would be lighter weight solutions with on-the-fly deployment and simple membership management.

Amazon's CloudWatch provides a generic and comprehensive solution for monitoring resources, applications, and services. It monitors metrics generated by a customer's applications, and it provides system-wide visibility into resource utilization, application performance, and operational health. However, its closed source nature restricts its use to Amazon web services and its cloud resources.

Project Hoover provides a simple way to collect metrics about the operational behavior of stream processing systems like those constructed with Flume. Specifically, we use Scribe and Pastry, which jointly provide efficient request reply routing and fault-recovery and can quickly adapt to the arrival and departure of nodes. Statistics from different flow paths are aggregated by publishing them in dedicated topics supported by Scribe, thus using its "push" model to gather statistics from nodes in each tier of the system. It then uses a "pull" model to aggregate the gathered statistics to form a global snapshot of the overall system. An external `aggregator' uses aggregate information to run models like those that evaluate and/or predict log traffic to adjust the number of collector machines in the system.

# 3. HOOVER'S DESIGN

As shown in Figure 2, the Project Hoover middleware has four main elements. They are (1) monitoring of local statistics for each agent and collector, (2) separately summarizing/publishing statistics within the agent group and the collector group, (3) merging the summarized statistics of two groups and sending it to an external aggregator using a Scribe anycast message, and (4) the ability to run online models that use summary statistics to dynamically tune the number of collectors in the system.

## 3.1 Local Node Metric Collection

Hoover models every node in the Flume system as an individual queue, and then, agents and collectors belonging to a particular data flow form a network of queues. Flume nodes expose three queue variables: input rate, output rate, and queue length. However, Flume's implementations of source and sink elements do not have built-in queues, thus requiring Hoover metric collection to emulate them, as explained next.

Flume's 'tail' source reads a single line from a file and converts it into an event. The events are then pushed out of a sink element. We interpret the number of lines read per unit time as the input rate of the queue; the number of lines that exit the sink per unit time is the output rate of the queue; and the average difference between the number of lines read and the number of events that exit the sink per unit time is the length of the queue.

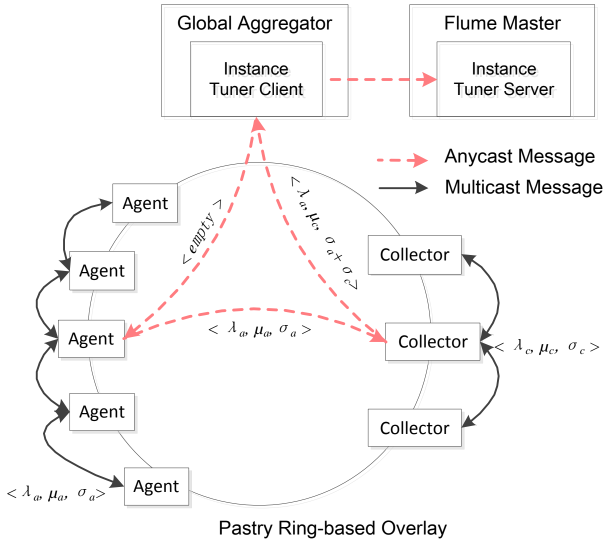


**Figure 2. Architecture of Hoover.**

Source and sink elements in logical nodes expose a Reporting object, which provides generic methods to add queue variables and other metrics as key-value pairs. The Reporting objects from the source and sink are refreshed during every heartbeat interval. Logical nodes then compute the Exponential Moving Average (EMA) of reporting objects to form a local snapshot. For other log processing systems, there may be alternative methods for defining queue variables. Hoover's performance modeling and auto-scaling methods will apply as long as such methods can be defined.

## 3.2 Local Metrics Aggregation

Hoover obtains scalability and fault tolerance by using Scribe as the communication substrate to exchange Flume metrics. Scribe is an application-level group communication system built upon Pastry, a DHT-based P2P overlay. Each pastry node is assigned a nodeId based on its IP address. Pastry routes messages to a given node by forwarding them to another node with nodeId numerically closest to the destination node. Every node maintains a small routing table with $O(\log_2 n)$ entries, which implies that Pastry can route messages in $O(\log_2 n)$ hops. Scribe nodes create and subscribe to multicast groups, called Topics. Subscribed members can publish messages to a topic, which will then be distributed in a multicast tree to all subscribed nodes. The root node of the multicast tree is the pastry node with nodeId closest to the topic name. A new node can subscribe to the topic by computing the key from the topic name and then using Pastry to route a subscribe message to the topic towards the root node. When a Pastry node receives a subscribe message from another node, it adds the node ID to its list of children and begins acting as a forwarder of the topic. If the Pastry node is already a member of the same group, it stops forwarding the subscribe message. Fault tolerance is achieved via timeouts and keep-alive messages. Specifically, if a child does not hear from its parent for some timeout period, it sends a new subscribe message to the root and is spliced to the multicast tree. When the parent does not hear from its child, it removes that node from its list of children. Scribe overcomes root node failure by moving it to the Pastry node with the next numerically closest nodeId to the key computed from the topic name.

Hoover uses Scribe multicast trees for Flume agents and collectors to aggregate and summarize their local snapshots. Every logical node spawns an instance of a Pastry node and a custom Scribe client application [12]. Local snapshots from agents and collectors are collected to calculate a group snapshot that contains the group's average input rate, average output rate, and average queue length. We create two aggregation trees, rooted at two rendezvous points, to disseminate local snapshots to other members. These two trees are the "AGENT" tree and the "COLLECTOR" tree. Agents subscribe to the "AGENT" tree, while collectors subscribe to the "COLLECTOR" tree.

As shown in Figure 3, each node has in its local snapshot local statistics stored as a set of (*attributeName*, *value*) pairs, such as (*InputRate*, 10). Periodically, each agent node triggers a multicast message and passes its local snapshot to every other node in the "AGENT" group. Other nodes save others' values in their respective caches. The same actions are taken by collectors. Over some slightly longer period of time, once one collector receives results from the "AGENT" group, it will combine them with the average of its caches and sends the final global snapshot to the external global aggregator.

If $t_n$ is the interval of time after which agents and collectors publish their local snapshots to other members of their respective Scribe groups, then the local snapshots contain the EMA of the queue variables between times $t_n$ and $t_{n-1}$. Each logical node resets its local snapshot and multicasts it to other members of the same Scribe group. Once all nodes have finished multicasting their respective local snapshots, every node now contains an in-memory cache of local snapshots of all members in its group. From this snapshot, it can then compute the average input rate, average output rate, and average queue length for the group, to form a group snapshot. By simply replicating that snapshot to all other group members, the same snapshot data is now available to all group members, regardless of node failures or removals.

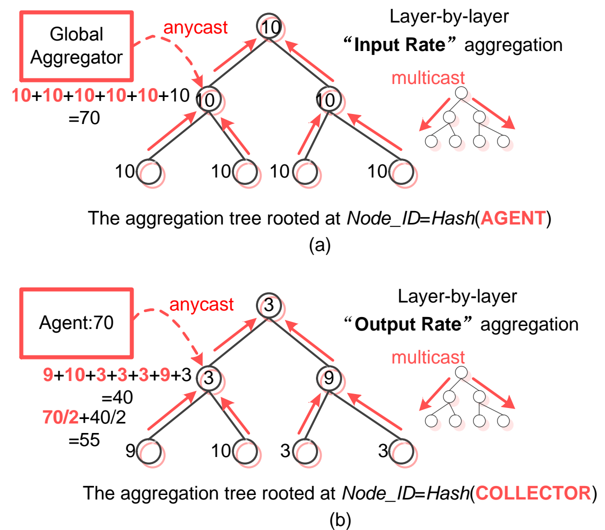


**Figure 3. Local Metrics Aggregation Trees.**

## 3.3 Global Metrics Exchange

Once the group snapshots have been computed by the agent and collector groups, they should be analyzed globally to determine the overall health of Flume components. Toward this end, the group snapshots from both the agent and collector groups are merged to give a single update vector of queue variables for the entire network of queues, called the global snapshot. The global snapshot is then passed to a regression model, which in turn uses them to predict the number of collectors required to handle the current aggregate volume of data.

We employ an independent node, called the aggregator, which is separated from the Flume system, to combine metrics and perform prediction. For example, let $m$ and $n$ be the number of agents and collectors in a data flow when it is polled at time $t_n$. Let the local queue statistics of every node be $< \lambda, \mu, \sigma >$, where $\lambda$, $\mu$, and $\sigma$ *is* the average input rate, average output rate, and average queue length of that node between two successive polling, $t_n - t_{n-1}$. The group snapshot of agent nodes is $< \lambda_a, \mu_a, \sigma_a >$, where $\lambda_a = \sum_i^m \lambda_i$ , $\mu_a = \sum_i^m \mu_i$ and $\sigma_a = \sum_i^m \sigma_i / m$ . Similarly, the group snapshot of collector nodes is $< \lambda_c, \mu_c, \sigma_c >$, where $\lambda_c = \sum_i^n \lambda_i, \mu_c = \sum_i^n \mu_i \text{ and } \sigma_c = \sum_i^n \sigma_i / n$. The merged global snapshot sent to the aggregator is $< \lambda_a, \mu_c, \sigma_a + \sigma_c >$. The snapshot of every logical node is adjusted to the configured polling period $p$ by linear interpolation or extrapolation, if the gap between two successive polling times, $t_n - t_{n-1} \neq p$.

A global metrics exchange is initiated using a single pull operation, which sends an anycast message to the agent group. One of the agents receives the anycast message and computes the group snapshot from its in-memory cache. The agent group snapshot is then anycasted to the collector group. One of the collectors computes its group snapshot and merges it with the agent group snapshot to form the global snapshot of the entire queuing network, and it then sends another anycast message to the aggregator with the global snapshot. Using anycast permits the aggregator to remain agnostic of Flume node failures and reconfiguration, reduces overall scribe message traffic, and avoids the need for procedures that manage global metric aggregation in lieu of node failures (e.g., assigning coordinators, dealing with failover, etc.).

## 3.4 Dynamic Instance Tuning

The aggregator applies the global snapshot collected during every polling period to a statistical analysis model used for auto-scaling. The experiments reported in this paper use the Secant root finding method to automatically scale the number of collectors based on the current health of the Flume subsystem. The intuition behind this method is that when the volume of events increases, the auto-scaler automatically adds more collectors to the system, thereby avoiding collector overload.

The health of the Flume system is assessed via two parameters, computed based on event output rate, event input rate and queue size. Let $\alpha$ be the percentage of events that leave the system with respect to the input rate. In our experiment, we set $\alpha$ to be 90, i.e. the Flume system is considered `healthy' if 90 percent of events have been processed by the collectors, $\mu_c / (90\% \times \lambda_a) \geq 1$. Similarly, let $\beta$ be the maximum percentage of input events that can wait in any of the queues in the Flume subsystem, i.e., if $\beta = 0.2$, then the Flume system is healthy if it is true that 99.8 percent of the events have exited the Flume system, $99.8\% \times (\sigma_a + \sigma_c) / \lambda_a \cong 0$. The overall health of Flume is denoted by $f$:

$$f = \frac{\mu_c}{\alpha\,\% \times \lambda_a} - \frac{(1-\beta\%) \times (\sigma_a + \sigma_c)}{\lambda_a} \tag{1}$$

When $f \cong 1$, the flume subsystem is healthy. The auto-scaler's health model computes $f$ only after a certain volume of global queue statistics snapshots has been collected, called a *window*. Every new computation of $f$ on a window $w$ initiates a new *phase* represented by $p$. Let $x_n$ and $x_{n-1}$ be the number of collectors in the Flume system during phases $p_n$ and $p_{n-1}$, respectively. Let $x_{n+1}$ be the number of collectors required for the next phase $p_{n+1}$. The number of collectors to deploy in the Flume system such that $f \cong 1$ after phase $p_n$ is the given by:

$$x_{n+1} = x_n + \big(1 - f(x_n)\big) \times \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \tag{2}$$

In order to make this method robust, the following constraints have also been applied.

- If $f(x_1) > \alpha$, then $x_{n+1} = x_n$.
- If $x_n = x_{n-1}$ and $f(x_1) < \alpha$, then $x_{n+1} = x_n + \gamma$, where $\gamma$ is a constant
- If $f(x_1) \gg \alpha$, then $x_{n+1} = x_n - \gamma$. This step also initiates scale-down when there is a larger number of collectors than is needed to handle some volume of input events.

Auto-scaling is implemented with an instance tuner composed of a Thrift client in the aggregator node and a Thrift server in the Flume Master. The Thrift server exposes interfaces to get and set the required number of collectors in the Flume system. The Flume Master is responsible for selectively activating and deactivating collectors when one of the clients sets a new instance count for a particular flow. The Master has pluggable Translation Managers, which transform or translate complex logical node configurations into compositions of simpler source and sink elements.

Flume already supports a Failover Translation manager that translates certain special source elements into a random list of collector nodes that are used as failover chains in logical nodes. These nodes are chosen from a master list of registered collectors. We have modified the Failover Translation Manager to support dynamic modification of the master list. The translation manager registers nodes with source element 'autoCollectorSource' into the master's collector list. If $x$ is the number of collector instances set by an Instance tuner client, then the translator picks up the first $x$ collectors from the master collectors list and places it into a Consistent Hash Table with replication. The key for each collector is computed based on its hostname and a random index number. The elements in the hash table are adjusted when the instance count changes. The sink element 'autoSink' is translated into a list of three collectors chosen from the hash table such that they are numerically closer to the hostname of the logical node being configured. Since the elements in the hash table are replicated with different key values, the data loads on the collectors are evenly distributed.

# 4. EXPERIMENTAL EVALUATION

Hoover is evaluated over a Flume system with 50 agents and 10 collectors. Every agent has a load generator program that writes events of a given size to a set of files in a designated folder at a given rate. The agents read these files as input events using the 'TailDir' source. Agents are configured with 'AutoBEChain', a

sink element which automatically switches to a different collector in case of failure, thus guaranteeing best effort delivery. Collector nodes are configured to send events to HBase which periodically writes the events to HDFS. Our setup used 6 HBase region servers and HDFS data nodes. A single aggregator is used to collect global statistics for the default flow. Global snapshots are analyzed by a regression module which predicts the expected number of collectors using the Secant root finding method to automatically scale the number of collectors based on the health of the flume subsystem, computed as explained above. The intuition behind this method is that when the volume of events increases, the auto-scaler automatically adds collectors to the system. The method is not described in detail because for this paper, our purpose is to demonstrate auto-scaling viability due to the monitoring and metric analysis capabilities of Project Hoover.

## 4.1 Aggregation and Round Trip Delays

The average aggregation time within a group is measured by recording the start time in the local Flume node and storing the difference between the receive time and start time in every receiver as part of the group snapshot. The group snapshot also records the number of local snapshots contained in it. The Round Trip Time (RTT) is calculated by recording the difference between the start time of the anycast probe message and the time when a global snapshot is received.

The following graph shows the average aggregation time and RTT in milliseconds within a group for a small volume of events, each of size *100 Bytes*, generated at the rate of *1000 events/second*. The scalability of the Pastry network can be verified by the results from [14]. Every Scribe node has a maximum of 6.2 children for a Pastry network size of 100,000 nodes. The aggregation time is in the order of tens of milliseconds. However, our current design uses a push model by multicasting updates to all members of the group. Hence, the number of local snapshots contained in a group snapshot shows some inconsistencies. This can be solved by using direct messaging instead of a group multicast, where each Flume node sends its local snapshot to its parent, which is then cascaded up to the root of the multicast tree [15]. The root can then multicast the final group snapshot to all other members.

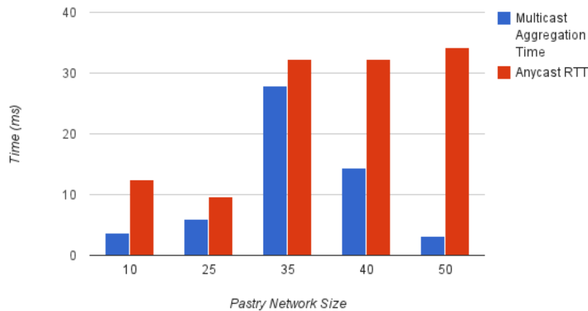


**Figure 4. Graph showing the average time taken for Multicast aggregation and Anycast aggregation.**

## 4.2 Auto-Scaling

In order to evaluate the utility of Project Hoover for online control, we use if for online Flume auto-scaling, evaluated with three independent runs with varying numbers of agents and collectors in the system. The appropriate number of collectors in the system is calculated in phases, say $n$. Initially, every flow is given a single collector to handle all of the agents. Global snapshots, health scores $f(x_n)$, current number of collectors $x_n$ and predicted number of collectors $x_{n+1}$ for the next phase are recorded. The graphs below show that our simple auto-scaler provides a good approximation for agent and collector combinations that have high average queue lengths and low health scores in the initial phases. With this behavior, a system with lower health scores is scaled much faster than a system with higher health scores.

Figure 5 shows how the auto-scaler increases the system's health scores gradually with varying numbers of agents and collectors. However, when pursuing a higher health score conflicts with the goal to also improve throughput, i.e., higher throughput, Project Hoover's auto-scaler will strike a balance between health score and system throughput. As shown in the yellow line of Figure 5, initially the system is regarded as quite healthy with 20 agents and 1 collector, $(f(x_n) = 0.99991741)$. However, the throughput is quite low for phase $p_1$, so the auto-scaler increases the number of collectors even through this then causes a decrease in health.

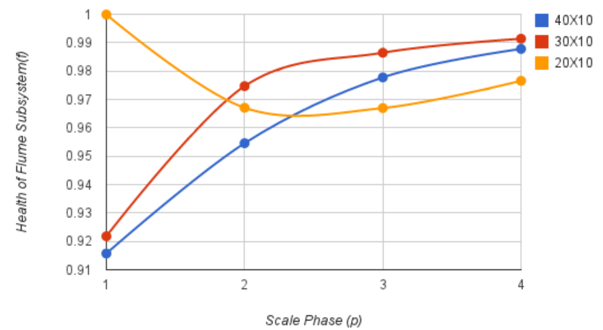


**Figure 5. Graph showing the overall health of the Flume System at the beginning of every phase.**

Figure 6 shows that the auto-scaler balances the average queue lengths of the system in order to obtain increased throughput. When new collectors are added to the system, the additional load will be distributed across a larger number of collectors, so that events that were previously buffered in the agents and collectors can be processed more quickly, thereby improving Flume's overall health.

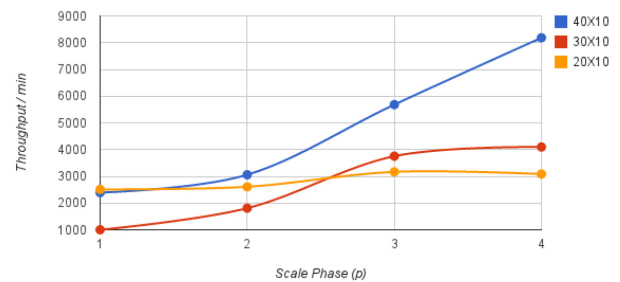


**Figure 6. Graph showing drastic increase in throughput for Flume systems with low health scores. The throughput of a Flume System with good health scores increases gradually.**

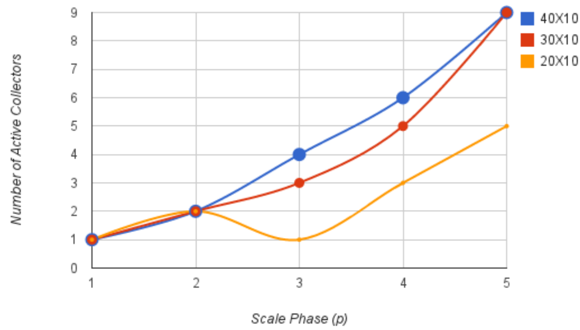


**Figure 7. Graph showing the number of active collectors at the beginning of every phase. The last reading in the graph shows the predicted number of collectors for phase $p_5$ to be added or removed at the end of $p_4$.**

Figure 8 shows that the average queue length of the system decreases as the predicted number of collectors is added to the system at the end of each phase. The yellow line in Figure 8 shows an increase in queue length at the beginning of phase $p_3$. This instability is reflected in the overall health of the system at the beginning of phase $p_4$, as shown in Figure 5. In response, the auto-scaler adds two more collectors to the system, as shown by Figure 7. This improves the system's health and also decreases the average queue length.

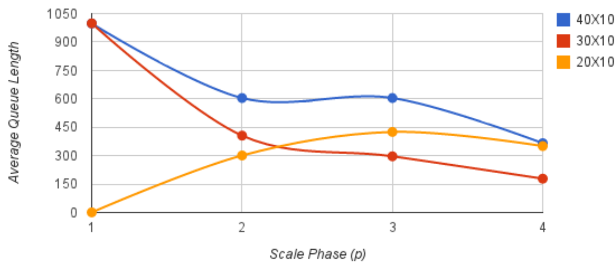


**Figure 8. Graph showing that average queue length of the system at the beginning of every phase.**

## 5. CONCLUSIONS AND FUTURE WORK

Project Hoover is scalable monitoring and aggregation middleware for event collection and aggregation systems like Flume. Its utility is demonstrated with methods that automatically right-scale the number of Flume collectors in the system depending on evaluations of overall system health. The auto-scaler models every Flume node as a queue and computes queue statistics such as average input/output rates and average queue length.

Project Hoover benefits the Flume log processing system because it permits the implementation of auto-scaling methods that can prevent the volume of input logs from overwhelming the intermediate processing nodes, called 'collectors', thereby maintaining the overall health of the Flume OG. Hoover's simple design can be realized in a scalable fashion, without requiring changes to the applications being used or to underling data center hardware/software. It has three unique characteristics:

First, Hoover uses a publish/subscribe based multicast and anycast mechanism deployed separately from the application (i.e., Flume in this paper), which aggregates local queue statistics to obtain a global snapshot of those statistics. Separation opens the door to deploying Hoover with a variety of datacenter codes, which we will demonstrate in our future work.

Second, Hoover has a dedicated component, called Aggregator, which periodically polls the members of the flow using anycast messages and gathers global snapshot data. Performance evaluations measuring aggregation delays within a group and across groups show that the approach is scalable for large group sizes.

Hoover's framework is fully implemented, but additional work is required to better predict the number of active collectors for varying loads. This includes (1) analyzing the distribution of real live logs generation, e.g., Poisson or Normal distributions, and simulating it with a log generator; (2) mature queuing theory can be employed for better predicting the number of collectors; (3) additional methods are needed for load balancing collectors, particularly when they perform computationally expensive tasks.

## 6. REFERENCES


[1] Amazon Cloudwatch. http://aws.amazon.com/cloudwatch/.

[2] Cloudera Flume. https://github.com/cloudera/flume/.

[3] Facebook Scribe. https://github.com/facebook/scribe/.

[4] Ganglia Monitoring System. http://ganglia.sourceforge.net/.

[5] HBase. http://hbase.apache.org/.

[6] HDFS. http://hadoop.apache.org/hdfs/.

[7] Jolokia. http://www.jolokia.org/.

[8] Java Management Extensions (JMX) Technology. http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html/.

[9] Remote Method Invocation. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html/.

[10] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa: A Large Scale Monitoring System. *Cloud Computing And Its Applications*, 2008.

[11] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, SCRIBE: A large-scale and decentralised application-level multicast infrastructure, *IEEE Journal on Selected Areas in Communication* (JSAC), Vol. 20, No, 8, October 2002.

[12] L. Hu, K. D. Ryu, D. D. Silva, K. Schwan. v-Bundle: Flexible Resource Offerings in Clouds. *In Proceedings of the 32nd IEEE ICDCS Conference*, Macau, China, June 2012.

[13] J. Kreps, N. Narkhede, J. Rao. Kafka: A Distributed Messaging System for Log Processing. *NetDB workshop*, 2011.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms* (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.

[15] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, M. Wolf. "A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-Scale Data Centers." In Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011), June, 2011.