Massively Distributed Monitoring

Christopher Peplin (peplin@cmu.edu)

May 1, 2011

Abstract

As the scale of distributed systems continue to grow, the basic question of monitoring the system's status becomes more difficult. How do you monitor an extremely large scale network of nodes without requiring a massive, centralized cluster for data collection?

To monitor these systems from a central location would mean potentially hundreds of thousands of new connections every second, each with a very small update. Persistent connections do not help, since a server cannot maintain that many open connections.

This paper evaluates different approaches to monitoring and describes the implementation of a few novel monitoring techniques in the peer-to-peer video distribution network Astral. Peers in Astral use a self-organized dynamic hierarchy, temporal batching and update filtering to increase the scalability of the monitoring subsystem. Some of the trade-offs associated with these optimizations are also enumerated.

1 Introduction

Computer system architecture shifted three major times in the past 30 years. Individual, personal computers gave way to mainframes and thin clients. The tide shifted back to powerful end-user clients in the 1990's. Now, the cloud and web services embody somewhat of a throwback to the mainframe era.

A web service is both distributed and centralized — distributed in that the client and server are separated; centralized in that all clients of a particular service connect to the same central hub. In data centers and a few emerging peer-to-peer applications, smaller distributed systems are emerging. The data center ecosystem itself is one massively distributed system, encompassing thousands of nodes across a diverse geography. In short, distributed systems are more common than ever and the importance of monitoring, tracking and accounting hasn't waned.

1.1 Problem

As distributed systems become the norm, business process managers are more and more interested in collecting knowledge of the behavior of the system. Planning and marketing are also increasingly data-driven. Each client, server or peer in a system generates potentially valuable usage statistics. System operators want to access this data from the granularity of an individual node up to an aggregate value for the entire system, or a value derived from many statistics.

As the scale of a distributed system increases, the monitoring task can potentially become a burden for an application. The demands and overhead of monitoring can equal or exceed those of the system's normal duties.

A prime example of a problematic monitoring situation is that for large peer-to-peer networks. With lessons learned from the centralization of Napster and the unstructured overlay network of Gnutella, modern peer-to-peer applications like BitTorrent focus on a completely decentralized operation that incorporates a minimal amount of local structure in the network graph. BitTorrent trackers can collect statistics on the files they seed, but statistics for a resource distributed among multiple trackers are difficult to reconcile.

Another example is massively multiplayer online games. Blizzard has a vested interested in tracking the users of their massively multiplayer online game World of Warcraft for billing, game balancing and to plan for future expansions. In order to scale the game world in a reasonable fashion, the developers split up the environment into thousands of shards. The gameplay statistics must make it back to a centralized location at Blizzard eventually, but the fractured architecture doesn't lend itself to a simple solution.

Civil infrastructure monitoring efforts have also encountered these issues. Consider a thousand mile gas pipeline — current monitoring approaches require wireless sensors spaced evenly along the route, which can quickly outpace a large data center in raw number of nodes.

1.2 Research Goals

This paper summarizes the challenges with large-scale, distributed monitoring systems and details some accepted solutions and their limitations. To test these ideas, we extended the logging and monitoring functionality of the experimental distributed system Astral [12].

Astral is a peer-to-peer streaming media content delivery network. The intent of Astral is to leverage the available upstream bandwidth of users watching a live video stream to alleviate the stress (and bandwidth bill) of the stream provider. Instead of all retrieving the video stream from a central location, clients look among network peers for those watching the same event. To match the quality and quantity of metrics available from a centralized system (and demanded by management), Astral must provide detailed usage statistics on the streams and their viewers. Using Astral, we found some of the limits of centralized data collection and tested the feasibility of a more distributed approach.

Definitions

- Source / Node — The root source of a statistic, e.g. a user's client connected to the Astral network.
- Sink / Collector — The hub for collecting statistics from sources.
- Supernode — A parent node in charge of 1 to n child nodes. The leader of a neighborhood of nodes. Typically not specially deployed hardware, but common nodes promoted by the system dynamically.

2 Monitoring

2.1 Metrics

The range of specific metrics that a system designer or operator might like to have is quite wide, and thus a monitoring framework must have a generic, unified interface for specifying the type, number and value of data points. It must be simple to add new metrics for each node, of different types of data.

In the context of an individual node, metrics include:

- CPU usage (instantaneous and averaged)
- Hard disk usage
- Memory usage
- Swap space usage
- Clock skew
- Network throughput
- Network latency

One level higher, applications have their own (potentially more interesting) metrics. For some common infrastructure-level components, metrics include:

- Database query latency
- Database index performance
- Database replication status
- Task queue failure rates

A peer-to-peer network's metrics include:

- Number of peers
- Supernode organization
- Supernode history
- Network membership history for a peer
- Specific data requested from the network
- Resource available at a peer

The metrics for an online game include:

- Individual player playtime
- Aggregate player activity
- Inter-player transactions
- Non-player character spawns
- Occurrence of in-game events

The types of civil infrastructure vary widely, but they tend to have close relation to a physical world element and external sensor input. Some sensor data may come in the form of video or images, further complication collection and storage methods. Common infrastructure metrics include:

- Temperature
- Pressure
- Flood gate status (boolean)

2.2 Challenges

The world of data center monitoring is changing rapidly. The developers of the popular monitoring tool Ganglia remark that "high performance systems today have sharply diverged from the monolithic machines of the past and now face the same set of challenges as that of large-scale distributed systems" [9].

The variety of systems described earlier are all beginning to look like data centers, and vice versa. The core goal of any monitoring system is a global view of the system for the purposes of health monitoring, performance optimization and accounting. An aggregate global view is more useful in a truly large scale system, where individual node failures are likely masked or handled by efficient failover. Consider that "failure" in a network of cable TV set-top boxes could be a user turning off a power strip each night.

The Ganglia developers suggest that the most important design challenges for a distributed monitoring system are scalability, robustness, extensibility, manageability, portability and overhead. This paper covers three of these in greater detail.

2.2.1 Overhead

Performance overhead can manifest itself on individual nodes or across the network as a whole. The monitoring system must not have a significant effect on the core task of the application, which means it must not consume significant CPU time, perform much disk access, or transfer large amounts of data over the network.

A small network footprint also lends itself to a more dynamic system. The less data that must be transferred, the quicker the operator can view the status of the entire system. Processing data as close to its source as possible can both lessen the network and central collection server load [3]. This must not come at the expense of application performance.

What exactly can and should be processed at the point of collection isn't always clear. For aggregate statistics, processing isn't possible without a full (or at least somewhat broad) view of the system. Nodes in a sensor network, oppositely, are able to process raw sensor data into a time averaged, human-parseable statistic before sending. In a general sense, the local processing optimization implies that data should be summarized when and where applicable before being sent to the sink.

2.2.2 Scaling

There are three primary techniques for scaling monitoring systems: hierarchical aggregation, arithmetic filtering and temporal batching. Unfortunately, all three introduce complexity, uncertainty and or delay and can make the system highly sensitive to failure [7].

Hierarchical Aggregation One problem for a centralized data sink is the sheer number of updates. A way to alleviate this stress is to aggregate the data from multiple nodes at strategic points in the network hierarchy. The exact size of each aggregated group is configurable, depending the desired load on the collection servers. The collected statistics can either be forwarded along as a batch of individual updates or combined into a single summary value (e.g. average CPU load across a cluster of nodes).

Unfortunately, network failures are amplified in a system with hierarchical aggregation. For example, "if a non-leaf node fails, then the entire subtree rooted at that node can be affected. [The] failure of a level-3 node in a degree-8 aggregation tree can interrupt updates from 512 leaf node sensors." [7]

Arithmetic Filtering Many metrics change infrequently, so arithmetic filtering can be used to limit the update frequency. After being reported to the sink once, the metric is cached and assumed to remain constant if no further updates are received. This only works for certain classes of statistics (boolean states are a good example), and also introduces ambiguity — it's difficult or impossible to distinguish between a non-reporting node that truly has a constant value and one that has failed [7]. One possible solution for identifying truly failed nodes is to use the existence of other updates from a node as an implicit aliveness update.

Temporal Batching For statistics that change frequently, but aren't immediately required by the system operator, temporal batching can further alleviate stress on the monitoring system. Either at individual nodes or combined with hierarchical aggregation, the values for a metric over a period of time are batched before being sent to the sink.

Beyond problems associated with the inherent delay in updates, temporal batching makes the system much more vulnerable to networking problems — "a short disruption can block a large batch of updates" [7]. Updates can be persisted to a log on disk at each node to make sure they are not lost.

A derived metric called network imprecision (NI) was proposed [7] to account for these variances when viewing system-wide statistics. NI is a "stability flag" that indicates if the underlying network organization is stable, which is a good indicator of the general accuracy of the statistics. To calculate NI, each update from a neighborhood of nodes must include:

- The number of nodes who may not be included in this update
- The number of nodes who may be double counted
- The total number of nodes

This solution for scalability introduces its own scalability issue — the system must report when nodes no longer are reachable, so an accurate accounting of the number of active nodes in the system is required. This will not scale well to a large peer-to-peer network without hierarchical organization and thus, we're back where we started.

2.2.3 Manageability

Manageability deals with both the system's own automated organization as well as that of the humans ultimately consuming the monitoring data. The management overhead must scale slowly with the number of nodes for the system to remain useful.

The management and monitoring tools of large distributed systems have the potential to become more complicated than the applications themselves. Components of the system can be grouped into organizations and split up work in a federated style to alleviate the management stress. The Domain Name System (DNS), for example, operates across thousands of administrative and technical domains by leaving many decisions up to the local administrators.

The Astrolabe [17] project proposed achieving scalability through a hierarchy of zones, each zone consisting of one or more nodes. The zone summarizes statistics into fields of a bounded size — e.g. a count of nodes with a certain property, not a list of their names. The system provides eventual consistency of these aggregate values, which is likely sufficient for many distributed monitoring tasks (especially considering the length of time it takes any action to propagate through an extremely large and diverse network) [17].

3 State of the Art

As data centers operations matured, monitoring coalesced around a few state of the art tools. This section describes some of the design decisions of these tools.

3.1 Storage

Any monitoring activity can generate a significant amount of data, given enough nodes and metrics. Even a single metric, measured often can quickly cripple flat-file storage and traditional databases.

As an evolution from flat-file storage, the designers of CARD [2] choose to use a standard SQL relational database. The motivation was the existing robust query language (SQL), and the ability to modify table definitions after creation. The developers were satisfied with their choice but there are a few issues for a larger scale system. Monitoring data is not especially relational and could better fit in a database intended for simpler data models. A database with map-reduce style querying could also provide a more natural interface. In fact, the designers had to add custom SQL syntax to allow flexible enough querying.

Every monitoring task is going to require this flexibility in data schema and query capability. A database specialized for such flexibility, specifically one of the newer schema-less databases like MongoDB [10] or Redis [16], is likely a better fit. Non-traditional databases are not new to monitoring — by far the most popular choice, used by both Ganglia and collectd [4], is RRDtool [1]. RRDtool is a circular database designed specifically for time slice data like monitoring statistics. The operator configures a maximum database size and older data is automatically overwritten in a round-robin fashion to maintain this size.

Operators who wish to archive such data must workaround its cyclical nature by moving weekly or monthly aggregate data to other RRDtool databases, and again to yearly and beyond. These quirks (and its rather slow performance when drawing graphs) make it a sufficient but less than perfect choice.

Finally, CARD only scaled to a few hundred nodes and due to their scale-up (and not scale-out) design, a relational database would likely have a difficult time keeping up if the system were to grow into tens of thousands of nodes.

3.2 Collection

The method a system uses for collecting monitored statistics can have a huge impact on its scalability; the two ends of the spectrum are push and pull.

Pull A pull approach requires the sink to explicitly request each desired data point from the nodes in the system. The sink must poll the nodes at some regular interval if continuous updates are desired. This requires central coordination and registration of all nodes in the system, which is often infeasible, especially in loosely structured peer-to-peer networks.

If the time it takes the sink to cycle through all of the nodes exceeds the polling time, time between polling must be increased (thus losing freshness). Parallel querying can improve this, but only up to a certain point, after which the pull model runs into an issue shared with the push model. Regardless of how often polling can be occur, it is likely that much of the data is duplicated or unchanged. This results in more network traffic than is necessary for freshness.

The pull method is used by the open source monitoring tool Munin, which periodically polls pre-registered nodes to retrieve updates.

Push A push approach places the burden of sending updates on the end nodes — either sporadically or at a regular interval, they send their updates to the sink identified by a known address. One problem with this approach is management and configuration. The update frequency and sink address are potentially difficult to change as they are distributed among many nodes (possibly not controlled by the operator). The use of a long-lived name and DNS can alleviate address changes, but updating any other settings will require that the nodes occasionally contact a configuration server to synchronize.

The scaling challenges of the push model are very similar to those of running a large web application — the sink must be able to handle a high request throughput, most of them very small write operations. The write-heavy workload is somewhat unique, but not especially challenging for today's databases.

The push method is used by two prominent monitoring tools in the industry, Ganglia and collectd.

Hybrid A hybrid push/pull gathering style can potentially minimize the duplication of data, maximum freshness and reduce network traffic. From cold start, the sink sends a request for data to each node with an associated count c. The node performs as in the push model c times, at which point the sink refreshes the query for another period. This allows occasional configuration changes to happen more naturally, while not introducing constant network overhead [2]. The nodes should self-register as in the push model.

At the time of writing this report, there were no widely used systems in the industry that use this hybrid approach.

3.3 Similarities to Civil Infrastructure

Interestingly, the shift in data centers to clusters of low to moderately powerful nodes brings the computing world more in line with the monitoring situation in civil infrastructure. Civil engineers and government organizations in charge of projects such as roads, bridges, oil & gas pipelines and waterways have been struggling with monitoring some of the earliest distributed systems. These are not distributed in the familiar computing sense; they are often entirely offline and unpowered. For decades, their data has been gathered (often inconsistently) by hand. The engineers tasked with accounting for trillions of dollars of public assets and physical systems are dealing with what could be viewed as widespread network unreliability and wholly unreliable nodes.

The increasing affordability of small, accurate, low-power sensors greatly interests civil engineers and infrastructure planners as the promise of accurate, fresh data is now realistic. Nearly all new construction comes complete with a range of sensors and a suite of software for monitoring and analyzing the current (and predicted future) state of the piece of infrastructure.

Modern research prefers wireless sensors over wired [8]. These systems are harder to disrupt, which is of greater concern when the system is out in the open and many network hops away from the central office. A widely distributed wired network involves a significant amount of extra infrastructure, and damage to the infrastructure being monitoring often implies damage to the monitoring system itself. Wireless systems have the advantage of being easier to deploy, as they can self-organize into ad-hoc networks as long as they are within range of another sensor node. Connectivity problems also tend to be isolated to individual units, and are easier to troubleshoot [8].

Recent projects have taken a page (knowingly or not) from tools like Ganglia and now use a unified data format, regardless of sensor or data type [8]. For example, a single update could consist of:

- Type of data (1 byte)
- Geographic coordinates, determined via GPS or inferred via signal strength of other nodes
- Network address
- Actual data (4 bytes)

These monitoring networks self-organize into a dynamic hierarchy of nodes based on their placement and capabilities. There are generally three types of nodes deployed:

- Basic sensor node
- Communication relay node — collects data in its 1- or 2-hop neighborhood
- Data Discharge Node — forward results to the Network Control Center, i.e. the one with a connection to the Internet

Whereas in computer system monitoring, each node generally has similar capabilities for communication and sensing, the role of these nodes are bound by their physical capabilities. Hierarchical organization becomes a simpler problem of guaranteeing a wide enough dispersal of communication relay and data discharge nodes to reach all of the leaves of the tree, compared to the somewhat arbitrary trees made in computer networks.

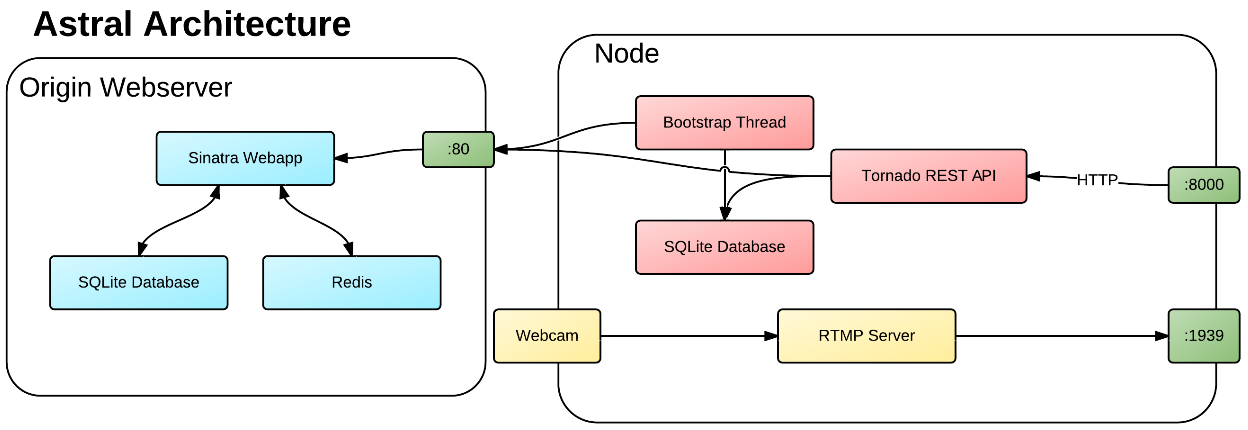

Figure 1: The major components of the central web application sink and a node in the Astral network. Nodes communicate with one another via HTTP using an embedded web server (the Tornado Python package). The web application that accepts statistics updates is written in Ruby using the Sinatra web framework. The system component for actually streaming video is completely separate - this is based around Adobe Flash's Real-Time Message Protocol (RTMP) [15].

4 Astral, a Testbed

Astral is a peer-to-peer content distribution network specifically built for live, streaming media. Without IP multicast, if content producers want to stream video of live events to users, they are forced to create a separate feed for each user. A peer-to-peer approach is more efficient and offloads much of the work from the origin servers to the edge nodes of the network.

Astral is built on the premise of having knowledge of a virtual overlay network of streaming clients. The system bootstraps itself and obtains this knowledge automatically through messaging among nodes and (to a limited extend) an origin web server. The nodes communicate with HTTP using an embedded web application running on each. They use simple JSON messages over the wire with a standard format for statistics. Each node runs a Python background process and the user sends control messages to it from the browser via simple HTTP requests in Javascript. Figure 1 illustrates the major components of both the nodes and the web application (which acts as both a coordination point and statistics sink).

Definitions

- Stream — A real-time data stream either of a live video source or of a stored recording
- Node — A networked computer running the Astral client and connected to the Astral network. The node can be acting as as producer, consumer, seeder, or a combination.

4.1 Goals

The primary motivation for Astral is the group project in Carnegie Mellon University's 18-842 Distributed Systems class [12]. A team of four developers (including myself) designed and implemented the peer discovery & organization protocol and streaming video service over the Spring 2011 semester. My own development efforts were also focused on adding a statistics generating and gathering component to the system for the purposes of this paper.

4.2 Challenges

In contrast to traditional file sharing peer-to-peer networks, Astral is purpose-built to distribute live media. This prompted some interesting design decisions; for instance, any client inside the network is guaranteed that what they are looking for is widely available. Simply a client's membership in the network is a hint that it has data to distribute to its peers.

Compared to a centralized distribution network, Astral's nodes must pay special attention to reliability. Users expect a steady video stream, even if the quality has to be occasionally reduced due to network congestion. In the centralized architecture, content producers provide reliability by scaling out with additional origin servers. In a peer-to-peer version, the departure of any one client could have a rippling effect on its peers. Astral keeps multiple streams open for the same content to increase robustness, similar to bonding multiple network interface cards together into one IP address.

Because of this resource duplication, the statistics component must take care to deduplicate stream popularity statistics. Each count must be identified with a unique node identifier to avoid counting the backup as well as the primary stream as separate users.

4.3 Statistics

The statistics monitored in the Astral network are:

- Current (deduplicated) number of nodes watching a stream
- Number of nodes acting as seeders for a stream
- Bitrate of the stream
- IP addresses of nodes watching a stream, for geographic visualization
- Network bandwidth of each node

4.4 Hierarchical Aggregation

The peers in Astral self-organize into a shallow hierarchy at startup, by going through this process:

- Query a central web application for a bootstrap list of supernodes
- Determine the round trip time to each supernode as a heuristic to find the closest
- Register with the supernode - the new node will be attached to this supernode for its lifetime. The relationship is stored persistently and survives peer restarts.

All statistics updates from peers are sent directly to their parent supernode, so beyond the initial bootstrapping step (which in total occurs only once per node) there is no load on the central server from individual peers.

The number of peers managed by a supernode is proportional to their available processing capacity, uptime and bandwidth. Long-living peers are obviously preferred to be supernodes to avoid requiring the re-registration of every child node. Each supernode can lessen the load on the sink linear with the number of child nodes registered with it.

4.5 Arithmetic Filtering

Astral performs limited arithmetic filtering for the stream viewer statistics. When a peer first requests a stream, that information is propagated back to the sink through a supernode. Once receiving the stream, the peer sends a heartbeat every 5 seconds to its parent supernode. The supernode does not propagate this back to the sink, and thus it assumes that the peer continues to watch the stream. When a peer leaves the network (either notifying the supernode during the proper shutdown procedure or as detected at the supernode by missed heartbeats), the supernode notifies the sink of the change in the peer's status.

This filtering minimizes the number of updates making it all the way back to the sink, but keeps the data as fresh as possible with what are essentially invalidation callbacks [6]. Without this filtering, the sink would have to manage the heartbeats, which could quickly overwhelm the server.

4.6 Temporal Batching

The stream provider is generally interested in the average bitrate of video received by the clients, but this information is not required to be completely fresh. Even after the live stream is concluded, this information is useful for provisioning network bandwidth in the future.

Astral takes advantage of this by batching 5 seconds of video bitrate statistics and returning only an average of these values to the sink. The batching is done at the level of individual peers, and in the future could also be performed at each supernode to further diminish the number of updates.

5 Evaluation

Astral is obviously a very young project, and quantitative analysis analysis at this point is likely to change quite a bit. However, we can do some basic comparisons between a baseline, completely centralized monitoring system and one with the various improvements discussed. These are currently mathematical projections, with the goal of performing practical tests when Astral's development settles.

5.1 Baseline Centralized

With a push-based collection style, the limits of a sink are very similar to that of a modern web application. The most common bottlenecks are:

- Throughput & concurrency capabilities of the front-end web server
- Throughput of the application server
- Performance of web application logic
- Database write throughput

A state-of-the-art web application stack geared towards a write-heavy workload could consist of:

- Nginx Web Server as the point of entry for requests [11]

Depth \ Batch Window Size	0	5s	10s	30s
One Level (Baseline Centralized)	100,000	20,000	10,000	3,333
Two Level (1000 supernodes)	1,000	200	100	33
Three Level (10 + 1000 Supernodes)	10	2	1	0.33

Figure 2: The effects of hierarchy depth and batch window delay on the overall number of update requests in a 100,000 node cluster. The cells contain the total number of requests per second received by the sink.

Depth \ Batch Window Size	0	5s	10s	30s
One Level (Baseline Centralized)	0	5s	10s	30s
Two Level	0 + 2 hops	10s	20s	60s
Three Level	0 + 3 hops	15s	30s	90s

Figure 3: The effects of hierarchy depth and batch window delay on the worst case freshness in a 100,000 node cluster. The cells contain the worst possible update delay in seconds.

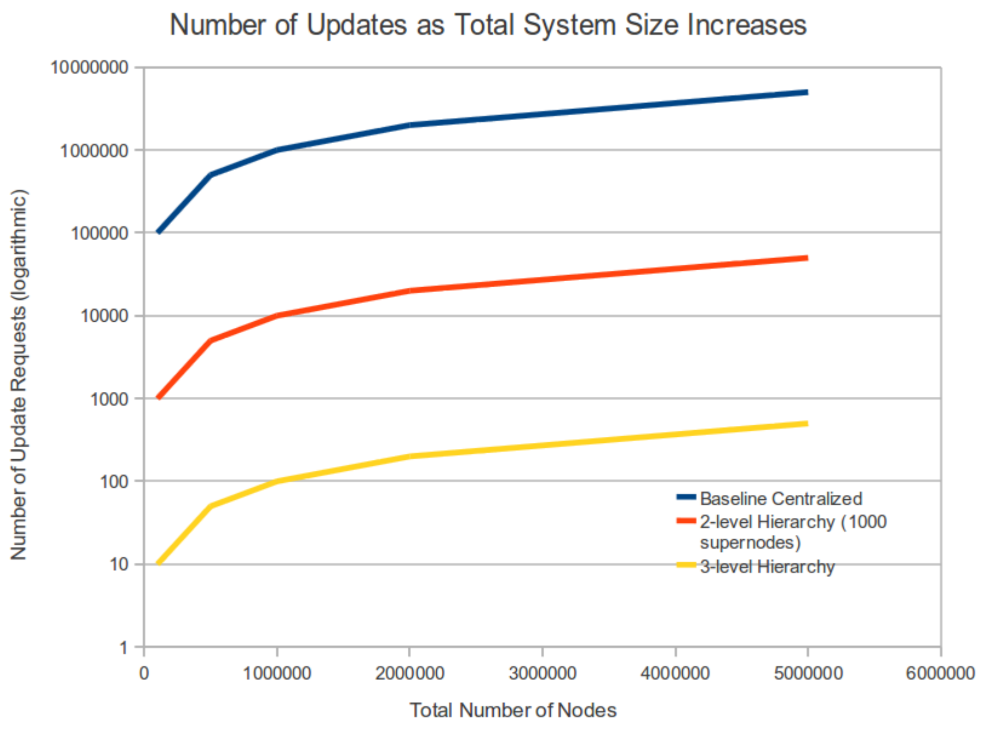

Figure 4: This graph relates the total number of nodes being monitored with the total number of requests per second that the sink cluster must be able to handle. The vertical axis is logarithmic to accommodate the large range of request rates.

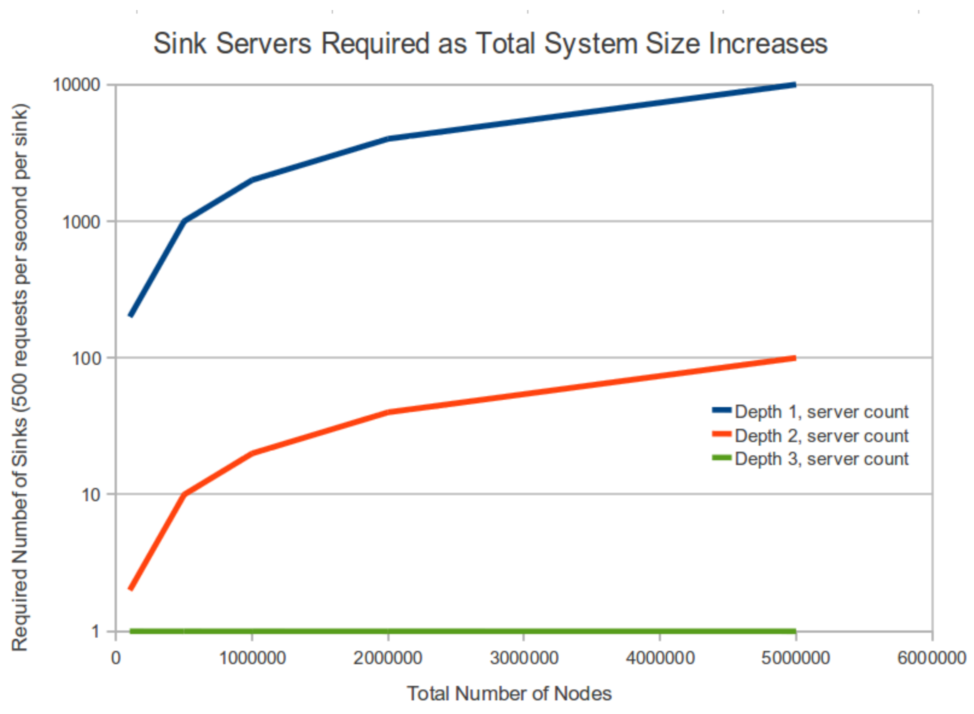

Figure 5: This graph relates the total number of nodes being monitored with the total number of sink servers required to process them, assuming a maximum average of 500 requests per second per sink. The vertical axis is logarithmic to accommodate the large range of request rates.

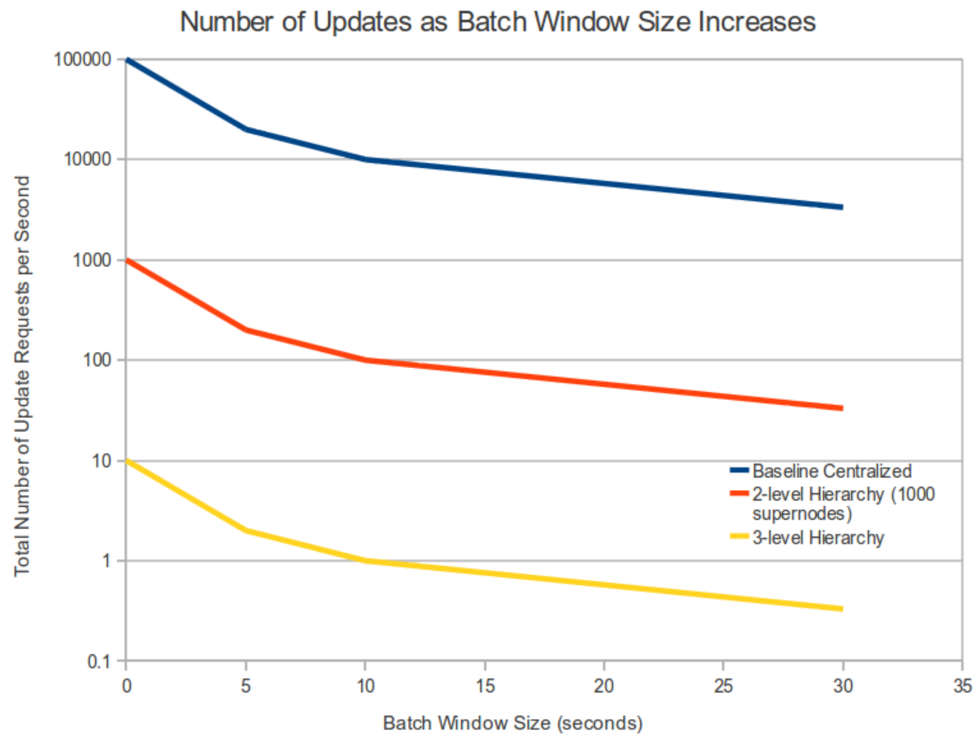

Figure 6: This graph relates the batch window size (in seconds) with the total number of requests per second, on average, that the sink cluster must be able to handle. The vertical axis is logarithmic to accommodate the large range of request rates.

- Phusion Passenger Ruby application server running 4+ concurrent OS threads [14]
- Ruby web application
- Redis, a high-performance key-value store [16]

The central web application component of Astral runs on this basic stack, deployed at the moment on the Heroku [5] platform. On an Intel Core 2 2.2GHz processor laptop with 4GB of RAM, the Redis database performs at an average of 40,000 set (i.e. write) operations per second.[1] A Ruby application in front of the database can server an average of 500 requests per second (and re-implementing the core statistics API in Java or Scala could increase that further) [13].

In order to have fresh information within 1 second, and assuming an average of 1 update per second from each node, the sink must be able to handle n requests per second, where n is the number of nodes in the system. With a cluster of four application servers and one Redis database, for example, the sink could handle updates from a 2,000 node system on average. The statistics in figures 2 and 3 for a 1-level hierarchy correspond to this baseline centralized case.

5.2 Optimized Distributed

A much larger scale system such as one for streaming the presidential inauguration (during which in 2009, CNN served 1.3 million concurrent streams at the peak) isn't feasible with this linear scaling factor. To handle that many nodes with a centralized sink would require a 2,600 server cluster just for monitoring.

The primary goal of the optimizations discussed in this paper is to lower the rate of updates from each node. Temporal batching like Astral's 5 second buffer lowers the rate by a factor of 5. Hierarchical aggregation lowers it by a factor proportional to the fanout of the supernodes. The effect of arithmetic filtering is more difficult to determine, as it depends on the length of time each node is connected. The longer a node is connected, the more the cost of the single connection request required is amortized. Short-lived clients will cost no more than with a non-filtered approach, and long-lived clients will avoid potentially thousands of requests over a one hour live event.

Figure 2 illustrates the effects of both tree depth and batching window size on the total number of update requests received by the sink. Something to keep in mind when adding levels to the hierarchy is the load on supernodes. If these are regular peers in the network, they may not be able to sustain a high rate of requests without impacting the user experience. These projections assume the supernodes are capable of an average of 100 requests per second, and is probably a bit high.

Figure 3 illustrates the worst case delay experienced due to batching. These projections assume that batching is done at every level (both supernodes and regular nodes). If it only occurs on individual nodes, the delay is never worse than that in a single level hierarchy (plus a negligible amount for the increased number of hops).

The delay from each of these optimizations can be predicted with some confidence. With a two-level hierarchy (a level of supernodes with regular nodes beneath each), the delay for batch updated is equal to the batching duration of each node. A 5 second window avoids a significant number of requests but doesn't significantly delay the data. Other monitoring tasks might be satisfied with even longer delays, up to minutes.[2] If the supernodes perform their own temporal batching, or there is a deeper hierarchy with additional batch windows, the worst case delay is only the sum of the batching windows along the height of the tree. Delay in one branch has no effect on the freshness of data from another.

A monitoring system with a two-level hierarchy and with a 5 second batching window on average (i.e. some may be delivered closer to real-time than others, based on the operators needs), the same four server sink cluster from the baseline centralized example could handle 1 million nodes (up from 2,000).

6 Conclusion

We have explored how many types of distributed systems are converging around very similar monitoring challenges, including traditional data center environments, civil infrastructure and peer-to-peer networks. The modern best practices described in this paper for scaling up a monitoring system to thousands of nodes come from the literature and existing systems in many fields and the experience of implementing Astral, a peer-to-peer content distribution network implemented in part to test out these ideas.

Astral as it stands is an incomplete system, and additional work is required before it is production-ready. Its performance and exact approach to monitoring will likely change. The quantitative evaluation

[1]Determined with the `redis-benchmark` utility, distributed with the Redis server package.

[2]Note that temporal batching could also be incorporated into a simple centralized collection architecture, with the same benefits.

of Astral can be extended in future work to determine the optimal values for configurable parameters such as the supernode fanout and batch window.

Monitoring these large systems is an increasingly large task, one that cannot take continue to take lower priority over other application features. Applications stand to benefit from decreased overhead if monitoring can be worked into the system early on in its development, and developers are encouraged to plan the accessible views into their systems as early as possible.

Source Code The Astral project is available as an open source project at `http://github.com/peplin/astral`.

6.1 Acknowledgements

Thanks to the students in the 18-845 course of Spring 2011 at Carnegie Mellon University who kindly review this paper and offered their feedback. Thanks also to Professor David O'Hallaron and Kushal Dalmia for their help with research for the project over the semester.

References

[1] *About RRDtool.* Apr. 19, 2011. URL: `http://www.mrtg.org/rrdtool/`.

[2] Eric Anderson and Dave Patterson. "Extensible, Scalable Monitoring for Clusters of Computers." In: *Proceedings of the 11th USENIX conference on System administration.* LISA '97. San Diego, California: USENIX Association, Oct. 1997, pp. 9–16. URL: `http://portal.acm.org/citation.cfm?id=1037150.1037153`.

[3] A. Asgari et al. "A scalable real-time monitoring system for supporting traffic engineering." In: *IP Operations and Management, 2002 IEEE Workshop on.* Dec. 2002, pp. 202–207. DOI: `10.1109/IPOM.2002.1045780`.

[4] *collectd - The System statistics collection daemon.* Apr. 19, 2011. URL: `http://collectd.org/`.

[5] *Heroku.* Apr. 19, 2011. URL: `http://www.heroku.com/`.

[6] John H. Howard. "On Overview of the Andrew File System." In: *USENIX Winter.* 1988, pp. 23–26. URL: `http://dblp.uni-trier.de/db/conf/usenix/usenix_wi88.html#Howard88`.

[7] N. Jain et al. "Network Imprecision: A New Consistency Metric for Scalable Monitoring." In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* UT Austin. 2008.

[8] I. Jawhar, N. Mohamed, and K. Shuaib. "A framework for pipeline infrastructure monitoring using wireless sensor networks." In: *Wireless Telecommunications Symposium, 2007. WTS 2007.* Apr. 2007, pp. 1–7. DOI: `10.1109/WTS.2007.4563333`.

[9] Matthew L. Massie, Brent N. Chun, and David E. Culler. "The ganglia distributed monitoring system: design, implementation, and experience." In: *Parallel Computing* 30.7 (2004), pp. 817–840. ISSN: 0167-8191. DOI: `DOI: 10.1016/j.parco.2004.04.001`. URL: `http://www.sciencedirect.com/science/article/B6V12-4CMHWWX-2/2/b6b44ba67c732867d1c3881c510b2953`.

[10] *MongoDB.* 10gen. Apr. 19, 2011. URL: `http://www.mongodb.org/`.

[11] *Nginx News.* Apr. 19, 2011. URL: `http://nginx.org/`.

[12] Christopher Peplin et al. *Astral.* Apr. 2011. URL: `http://astral.rhubarbtech.com`.

[13] *Performance and memory usage comparisons.* Ruby Enterprise Edition. Apr. 19, 2011. URL: `http://www.rubyenterpriseedition.com/comparisons.html`.

[14] *Phusion Passenger.* Apr. 19, 2011. URL: `http://www.modrails.com/`.

[15] *Real-Time Messaging Protocol (RTMP) specification.* Adobe. Apr. 19, 2011. URL: `http://www.adobe.com/devnet/rtmp.html`.

[16] *Redis.* Apr. 19, 2011. URL: `http://redis.io/`.

[17] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining." In: *ACM Trans. Comput. Syst.* 21.2 (2003), pp. 164–206. ISSN: 0734-2071. DOI: `http://doi.acm.org/10.1145/762483.762485`.