Scaling a Monitoring Infrastructure for the Akamai Network

Thomas Repantis
Akamai Technologies
Cambridge, MA 02142
trepanti@akamai.com

Jeff Cohen
Akamai Technologies
Cambridge, MA 02142
jecohen@akamai.com

Scott Smith
Formerly of Akamai Technologies
Cambridge, MA 02142
scott@clustrix.com

Joel Wein
Akamai Technologies
Cambridge, MA 02142
jwein@akamai.com

ABSTRACT

We describe the design of, and experience with, *Query*, a monitoring system that supports the Akamai EdgePlatform. Query is a foundation of Akamai's approach to administering its distributed computing platform, allowing administrators, operations staff, developers, customers, and automated systems near real-time access to data about activity in Akamai's network. Users extract information regarding the current state of the network via a SQL-like interface. Versions of Query have been deployed since the inception of Akamai's platform, and it has scaled to support a distributed platform of 60,000+ servers, collecting over 200 gigabytes of data and answering over 30,000 queries approximately every 2 minutes.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications, Distributed databases

General Terms

Design, Experimentation, Performance

Keywords

Distributed Systems, Scalability, Monitoring, Stream Processing, Distributed Databases

1. INTRODUCTION

Akamai's large-scale, on-demand, distributed computing platform [9] offers a variety of services. These services range from live and on-demand media streaming to the delivery of static and dynamic HTTP content, and from the acceleration of Web and IP applications to high-availability storage and DNS. Supporting these services is a platform consisting of over 60,000 servers deployed in 70 countries within about 1,000 autonomous systems. Each server runs multiple applications. As a result, Akamai's platform consists of over 1 million distributed software components.

Due to the scale of the platform, its purely distributed architecture, and the service requirements necessary to meet customer needs, it is critical to have a reliable near real-time monitoring infrastructure. Data must be collected from the edge servers as fast as possible, to monitor the usage of the platform, detect anomalies, and troubleshoot problems. *Query* is a near real-time monitoring system developed at Akamai to address these needs. It is used by operations staff and administrators, software engineers, monitoring and measurement systems, and indirectly by customers to glean information regarding the current status of the Akamai platform and the performance of the services it provides.

Data in Query is accessed via SQL queries, which allows users a comprehensible, familiar, and precise way of describing the information they want. It also allows them to combine data from many tables in traditional ways. For example, the SQL query below could be issued to show the average RSS memory usage of processes running as user id 5000, grouped by process name and software version, and filtered to only include processes with an average memory usage over 100MB:

```
SELECT p.process_name,
       sv.version,
       avg(p.rss) avg_memory
FROM   processes p, software_versions sv
WHERE  p.machine_id = sv.machine_id
AND    p.uid=5000
GROUP BY 1,2
HAVING avg_memory > 100*1000000;
```

Scalability is an important design consideration in Query, as it publishes data from tens of thousands of machines, with the overall amount of published data growing steadily over time. Query handles its scalability challenges by having a hierarchical architecture that is largely highly distributed, but exhibits some aspects of a hybrid centralized/distributed design. We explain this architecture in Section 2, and the lessons we have learned from building and operating Query in Section 3. We describe some of the applications using Query in Section 4 and the performance it provides them in Section 5. We close by putting Query in the context of related work in Section 6 and present conclusions in Section 7.

2. QUERY ARCHITECTURE

2.1 Design Goals

As a monitoring system for an Internet-scale network, Query has several design goals, including scalability, low latency,

reliability and fault-tolerance, data consistency and data completeness, and providing an atomic snapshot of the Akamai network. In addition, the design of Query must integrate with our overall system design sensibilities, which involve the use of redundant and distributed commodity hardware as opposed to smaller numbers of higher-end special purpose servers [2]. As it is difficult to achieve all of these in one system, the design of such a system must make tradeoffs among these goals. We now discuss each goal of Query's design in more detail.

2.1.1 Scalability

Query needs to scale as both the amount of published data and the number of queries being issued grow continuously. The growth in the amount of published data is driven by growth in the number of machines in the network (both to scale mature services and initiate new services), the number of tables reported by old and new software components, and the size of these tables, some of the largest of which grow with the size of Akamai's customer base and the size of the network. Section 3 discusses techniques Query uses to address its scalability needs.

2.1.2 Latency

Data provided by Query needs to be fresh. Although the automated response to network faults does not rely on Query, the timely availability of recent monitoring data is important to many applications. For example, system anomalies must be detected quickly to enable a rapid investigation and response. This is not possible if data are substantially out of date. Not only do the data used to answer a query need to be recent, but that answer needs to come back promptly. It is unacceptable for an application or user to have to wait a long time to get results. This leads to two notions of latency in Query: *data latency* (how old are the data) and *query response time* (how long does it take to get an answer). Query uses caching to reduce both types of latency, as described in Section 3.1. It also uses redundant machines to reduce the load of queries on each machine, to reduce query response time.

2.1.3 Reliability

With over 60,000 machines, some are always down or unreachable. Thus, Query's design needs to handle machine and connectivity failures gracefully. We address failures of the query infrastructure with redundancy, whereas we address failures of edge machines using algorithms for deciding when to temporarily cease waiting for data from a particular edge server. Section 3.6 discusses how Query handles faults in more detail.

2.1.4 Eventual Consistency and Synchronization

Maintaining a consistent view of a network of over 60,000 machines is a significant challenge. Each Query Top-Level Aggregator (TLA), which is a server that aggregates data from across the network into tables, can see some portion (possibly all) of the Akamai network. Query must guarantee that if a row is published on a particular edge machine, then every TLA that can see that edge machine will eventually have that row. Furthermore, it must guarantee that the data a TLA has from an edge machine can never become older; that is, older data can be replaced with newer data, but not the other way around. Finally, each TLA should have an approximately synchronous snapshot of the network; no two rows of data that it contains should have been published too far apart in time. Keeping low latency and fault-tolerance while guaranteeing universal consistency and synchronization at all times would be prohibitively difficult. Therefore, data completeness and synchronization within a TLA as well as across TLAs are best-effort.

2.2 Distributed Architecture

Query has a hierarchical structure that allows it to scale. The Akamai network is organized into *clusters*, which are groups of machines in the same physical location. There are thousands clusters of widely varying sizes, totaling over 60,000 machines. In each cluster, Query runs on each machine to collect data from all software components on that machine that publish into Query. Within each cluster, a set of machines called *Cluster Proxies* collect all the data from the cluster and combine them. *Top Level Aggregators* (TLAs) combine data from all clusters into tables. The TLAs then send copies of their tables to machines called *SQL parsers*, which then receive queries and compute their answers. While the TLA and the SQL parser functions can be co-located on the same machines, separating them allows them to be scaled separately (as TLA resources scale primarily with network and table size, while SQL parsers scale primarily with request load). Additionally, to help TLAs cope with the data collection load, the function of applying table updates can be split across multiple machines known as TLA children. We discuss this in more detail in Section 3.5. Figure 1 shows the basic Query architecture.

Some applications are not interested in data from the entire network, so only certain TLAs can see all 60,000 machines. Others can see only a subset of the clusters that serve a common purpose. We call such a group of clusters a *subnetwork* and a TLA that can see the entire Akamai platform a *Mega TLA*. The smaller subnetworks are small enough that the same machine has enough resources to both collect data and provide answers to queries. We call such a machine a *TLA/SQL*.

For fault-tolerance and scalability, most of Akamai's systems are highly replicated, and Query is no exception. Multiple machines serve data for each subnetwork. If one SQL parser goes down, applications switch to using another intended for the same role. This happens transparently to the applications because they use DNS names to locate SQL parsers. If one TLA goes down, SQL parsers switch to another. If one Cluster Proxy goes down, TLAs switch to another. Finally, if an edge machine goes down, we omit rows from that machine from Query until the machine comes back up.

2.3 Data Collection

When a user or application issues a query to a SQL parser, the SQL parser checks if it has all the tables that query uses. If it does, it computes the answer immediately. Otherwise, it sends a request for any missing tables to the TLA. If the TLA doesn't have all the tables it needs, it requests the missing ones from all Cluster Proxies, which then request them from their respective clusters.

Upon receiving a request for a table, Query does not send

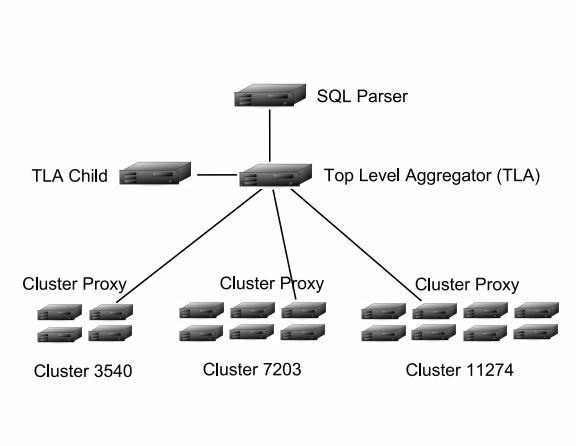

Figure 1: Basic query architecture.

the table back immediately, because that would violate our best-effort synchronization. Instead, it includes the newly requested table in its future nearly-synchronous snapshots. We call each nearly-synchronous snapshot a *generation*; generations start at predefined intervals (such as every two minutes), with some variation based on system load. (We discuss this in more detail in Section 5.)

Every generation, every edge machine sends to each Cluster Proxy all tables that the Cluster Proxy has requested, the Cluster Proxies combine the data they have received and send to each TLA all the tables that TLA has requested, and the TLAs send to the SQL parsers all tables that each SQL parser has requested. These sends are staggered such that each step happens shortly after the prior step completes. Once the SQL parser has all the tables, it answers the query. Because of temporal locality of reference, tables are likely to be reused again soon, so the SQL parser keeps requesting each table until some period of time has elapsed with no queries using it.

We know that some tables will be used frequently, so we speed this process up dramatically for all of those common cases by preemptively requesting those tables. We describe that process in Section 3.1.

Collecting data from a large number of machines exposes interesting tradeoffs between completeness and latency. For example, if a TLA has only received a copy of a table from half of the clusters, it could decide that it has the table, even though it is incomplete, or it could wait before sending that table to the SQL parsers. We choose completeness except in extreme cases, to maximize the visibility into the network. A Cluster Proxy, likewise, does not send a copy to the TLA unless all machines in the cluster that have the table have sent it. If an edge machine fails to provide a copy for three minutes, the Cluster Proxy will drop that machine's rows entirely. A TLA will likewise drop a cluster that fails to provide a table for three minutes after it is requested. A TLA does not provide a table unless all Cluster Proxies that have the table have sent their copies to the TLA.

Another motivation for this behavior is to provide clear semantics that rows reported directly by down machines or clusters will disappear after some period of time, thus simplifying the interface provided to the users of Query. Similarly, to provide clear semantics to Query's users, tables detailing the configuration of the network (such as the tables enumerating machines and clusters) are sourced, so as to be guaranteed to be provided in their entirety.

2.4 Aggregator Sets

Query's TLAs, TLA/SQLs, and SQL parsers (collectively referred to as *aggregators*) are divided into *aggregator sets* that serve a common purpose. Each aggregator set sees tables from a particular part of the Akamai network and is intended for certain users. That allows us to make sure that critical applications get the performance they need. For example, we do not want developers testing out queries to interfere with automated systems collecting important information used for operational monitoring of the network. We therefore have sets of aggregators specifically for certain applications or groups of applications. We control the load on critical aggregator sets to make sure they offer an acceptable level of service.

3. TECHNIQUES AND EXPERIENCES

Query is used widely within Akamai. This popularity, combined with application and network growth, has posed significant scalability challenges. In this section, we discuss the approaches we have developed and the lessons we have learned from building and operating this system. We describe the architectural and administrative techniques we have used to allow the system to continue to scale, and how we manage the system so that each class of users and applications can access the information they need with reliability guarantees appropriate to their requirements.

3.1 Caching, Prediction, and Compression Policies

Query employs a variety of policies to anticipate what data will be necessary to answer near-term future queries. In some sense, an aggregator contains the subset of global network data that constitute its best guess at what will be requested soon. It constructs this guess by requesting certain tables in advance (a process called prewarming), caching certain subqueries, and requesting updates for recently requested tables.

Each aggregator reads a configuration file (distributed dynamically through Akamai's configuration management system[11]) telling it what tables to prewarm. These tables are requested in each generation, even if no query has recently asked for them. As a result, as soon as a query arrives that wants to use only prewarmed tables, its answer can be computed immediately, ensuring that our most urgent queries complete quickly. Combined with the fact that a table that is not prewarmed is requested for some period of time after it is used (with that clock resetting after each later use), the vast majority of queries can complete without having to wait for Query to fetch tables.

Query caches the results of its views during each generation of tables, and its configuration files define views that describe many common sub-queries. This simplifies users'

SQL code and enables Query to compute the result of each view only once per cycle.

To minimize the communication related to collecting data from the edge, Query does not send full copies of every table. Instead, each cluster sends each TLA only the information about what has changed since the last generation. This saves about half the bandwidth Query would need to use otherwise. To further reduce bandwidth consumption, data are being compressed before being transmitted.

3.2 Network Partitioning

The Akamai network is not entirely homogeneous. For example, machines performing infrastructural tasks are often separated into different subnetworks from machines serving end-user requests. Often times, queries only need data from one of these types of machines. Because each subnetwork is somewhat smaller than the whole, an aggregator talking to only that subnetwork will have an easier time scaling than one combining data from across the entire network. Every subnetwork has such aggregators, and most queries use them. Only those queries that use the Mega aggregators, whose purpose is to join data from all subnetworks, experience the complete effects of our scale. Other techniques in this section are used to mitigate or eliminate those effects.

3.3 User Partitioning

Different users or applications have different requirements for Query. Some users need Query to handle a large volume of data, such as for analyzing traffic patterns across all machines. Other users need Query to handle as little data as possible, so that it can aggregate data and answer queries quickly with a high level of availability, so that Akamai operations can be alerted to anomalies requiring investigation. Some users need to test queries to understand the impact of their resource utilization on Query SQL parsers. These requirements are challenging to satisfy simultaneously.

We solve this problem by partitioning the use of Query. Depending on the volume of data and the load of queries we send to a TLA or SQL parser, it will have a certain speed of response and level of reliability. We divide up the load so that critical applications have their own dedicated aggregators, and no application runs against aggregators that fail to meet its reliability needs.

3.4 Table partitioning

In addition to partitioning based on subnetworks and usage, Query allows for table partitioning. Using this technique, aggregators for the same subnetwork and usage only serve a subset of tables, to reduce their load. Table partitioning significantly improves Query's performance and relies on two features of the usage pattern: (1) although SQL theoretically allows any set of tables to be joined, only certain sets can be joined to produce anything interesting; and (2) although the sets of tables that are used together can theoretically change at any time, in practice they change very slowly. Typically, we can divide the tables used by a particular user into several sets of approximately equal size. This can only go so far, as one of the partitions will still need to have all the tables used by the most resource-intensive query. But it means we can scale up to the point where a single query takes all of the resources of the machine. Indeed, a few Query machines are currently deployed to repeatedly answer a single resource-intensive query for a customer-facing application.

Partitioning is most useful if it can be done transparently to Query's users. Rather than pointing to specific Query machines, applications reference partitions in a manner that can be rapidly changed as partitions are reconfigured. Additionally, aggregators can send back messages redirecting users to other partitions, based on their view of how other aggregators are configured.

3.5 Clustering

Collecting, decoding, and combining data from over 60,000 machines and sending that to SQL parsers can be too heavy a task for a single TLA. Therefore, the load of a TLA can be shared among multiple machines: a TLA, and zero or more machines called TLA children. Each TLA can be told to distribute a certain set of tables among its TLA children by a configuration file, a process called deferring. Upon receiving tables from a cluster, the TLA will send the deferred tables to the TLA children, who will decode them, combine the copies from all the clusters, and encode them into the same efficient form the TLA uses to talk to SQL parsers. The TLA children then send back the encoded form of the table, and the TLA sends it to the SQL parsers along with tables from the TLA itself and from all other TLA children.

3.6 Handling Faults

With over 60,000 machines publishing into Query and over a hundred aggregators, some Query machines are down or unreachable at any time. In this section we discuss how Query copes with these failures. We also discuss what we have learned about tradeoffs between fault tolerance and fault isolation.

Query needs to handle failures of edge machines, dropped connections to edge machines, and aggregator failures. The latter are handled using redundancy. If a TLA goes down, its SQL parsers can switch to any of several others in the same aggregator set. If a SQL parser goes down, its users can still use others in the same aggregator set. Redundancy also helps address the failures of edge machines, since each cluster will have multiple Cluster Proxies. However, when an entire cluster of edge machines becomes unavailable, for example, due to a connectivity issue, aggregators temporarily drop it, instead of holding up new table updates waiting for its data or serving overly stale data.

Some of Query's features illustrate tradeoffs between fault tolerance and fault isolation. For example, fault tolerance dictates that if a SQL parser is talking to a TLA, and that TLA goes down, the SQL parser should switch to another TLA. Fault isolation, on the other hand, dictates that the SQL parser should be prevented from talking to another TLA, because the TLA may have gone down due to the size of the request from the SQL parser, in which case switching would just impact another TLA. By default, SQL parsers can switch to other TLAs in the same aggregator set. By changing configuration, SQL parsers can be rapidly reconfigured to only utilize a subset of the TLA set. That gives operations staff the flexibility needed to choose the best approach for every level of load and quality of availability needed.

4. APPLICATIONS

In this section, we describe some of the most significant users of Query.

One of the most important Query users is Akamai's "alert system", used to identify anomalies on the network. The alert system is based on SQL queries called "alerts" that are executed periodically. These alerts are defined by developers or operations engineers, and trigger based on violations of invariants (i.e., rows are returned by a SQL query) or when reported parameters cross thresholds. For example, an alert might indicate that a machine has been automatically taken out of service due to a hardware problem and needs repair. Alerts that trigger are responded to based on their priority and utilize associated procedures which define an appropriate response. This application depends upon several of Query's main design goals, including its near real-time view of the network and the completeness of the data it provides. The alerting system relies on high availability and low latency of data, but has load that remains fairly constant over time.

Another important Query user is a historical data collection system which is used for tracking quantitative metrics about Akamai systems over time and displaying them in graphs. Again, this system is based on user-defined queries that are executed periodically, and currently supports over 15,000 such queries. This system depends on a different set of features from the alert system. While latency is extremely important for alerts, historical data collection has lower latency requirements, counting instead on completeness, scalability (due to its high load), and eventual consistency. The graphs in Section 5 were collected with this system.

Finally, a variety of systems give access to Query data to Akamai's customers, including graphs available to the general public on the Akamai website [13]. These applications, often result in a lower system load than some other applications, but have a need for reliability, completeness, and correctness.

5. PERFORMANCE

In this section we present some of the interesting aspects of Query's performance. As the Akamai network has grown by multiple orders of magnitude, the techniques we described in Section 3 have allowed Query to keep data latency and response time within fairly constant bounds.

5.1 Data Latency

Data latency refers to how stale data collected from the edge are at the time they reach the aggregators. Figure 2 shows data latency over a one-day period for a heavily loaded Mega aggregator that spans the entire network. Data latency drops every three minutes, indicating that it takes the machine about three minutes to collect, decode, and combine data from all of the clusters of edge machines.

5.2 Query Response Time

Figure 3 shows the average elapsed time in milliseconds for queries issued to a Mega aggregator used by one particular application over an one-week period. This response time is the latency experienced by users or applications querying the system. It includes the time required to obtain the requested data and process any particular query. It also includes any queueing delays on the aggregator due to other concurrently executing queries. Thus, the response time depends on data availability and query complexity, but also on aggregator load.

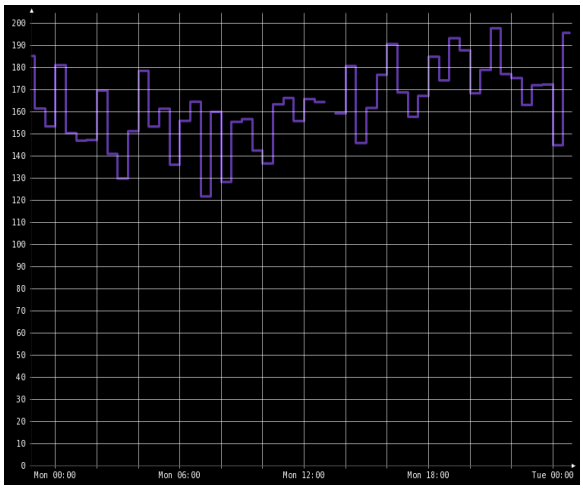

Figure 2: Data latency (in seconds) over a day on a heavily loaded aggregator.

One interesting observation about this graph is its apparent wave-like shape. As the total utilization of the Akamai network varies based on time-of-day, and because some query tables are sized proportionally to the current network utilization, aggregator load also varies based on time-of-day. The two lower peaks are for Saturday and Sunday.

6. RELATED WORK

Query has been in development for approximately 12 years. Rather than comparing Query today to the state of the art at its inception, in this section we reference related work that has appeared throughout Query's lifetime.

Large-scale network monitoring systems face challenges related both to data volume and network size, as well as network and machine failures. SDIMS [15] attacks the scalability challenges by using Distributed Hash Tables to create scalable aggregation trees. It also uses lazy and on-demand reaggregation to adjust to network and node reconfigurations.

PRISM [6] employs imprecision to tackle both scalability and failures. Arithmetic imprecision bounds numeric inaccuracy, temporal imprecision bounds update delays, and network imprecision bounds uncertainty due to network and node failures. Employing imprecision enables PRISM to reduce its monitoring overhead and yet provide consistency guarantees despite failures.

Several system administration tools such as Nagios [8] and SCOM [12] exist for monitoring network services and machine resources. Query serves a similar purpose, by allowing users to specify complex monitoring tasks using a SQL-like interface.

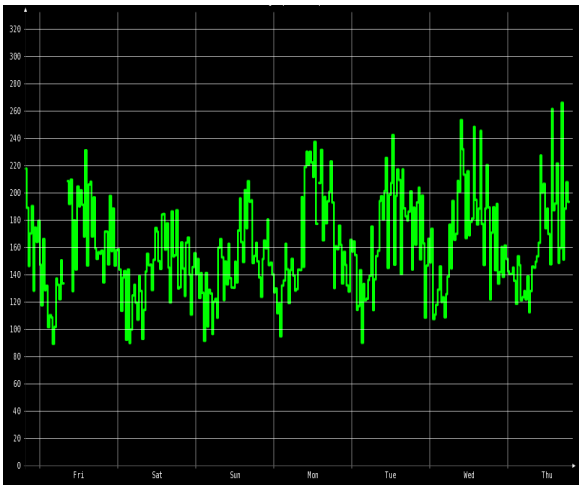

Figure 3: Query response time (in milliseconds) over a week. The oscillation is due to patterns of network usage. The size of many tables grows and shrinks with network load, which varies over the course of the day.

Real-time processing of high-volume, continuously updated data has been the focus of several research efforts in the area of stream processing systems. Telegraph [3], STREAM [7], and Aurora/Medusa [4] were the first generation of such systems, focusing on providing a SQL-like interface to query continuously updated data. Borealis [1], as a continuation of the Aurora Project focused on the challenges related to implementing such a system in a distributed fashion, with particular emphasis in load shedding and fault-tolerance. Synergy [10] has focused on composing distributed stream processing applications, while paying attention to their end-to-end Quality of Service requirements.

Distributed stream processing has been the focus of several industrial research efforts as well. IBM's System S [14] has focused on a variety of stream processing applications with highly variable rates, utilizing a large number of stream processing nodes. AT&T's Gigascope [5] has focused on monitoring network traffic at extremely high-volumes.

7. CONCLUSIONS

We have discussed the design, implementation, and lessons learned from operating Query, Akamai's large-scale monitoring system. Query enables the processing of data from over 60,000 edge servers in near real-time. The availability of a powerful SQL-like interface to that data has been an important element in how we manage our network. In this paper we have focused on the design choices that enable Query to scale as the network size, the data volume, and the number of queries grow. We have shown how Query addresses these scalability challenges, while providing a data latency in the order of minutes and an average query response time in the order of tenths of a second.

8. ACKNOWLEDGMENTS

The authors would like to thank everyone who has developed and supported Query throughout the years, including Andy Berkheimer, Josh Buresh-Oppenheim, Timo Burkard, James Chalfant, Ron Chaney, Greg Fletcher, Stephen Gildea, Dan Katz, Sef Kloninger, Alan Krantz, Phil Lisiecki, Andrew Logan, Brian Mancuso, Erik Nygren, Tim Olsen, James Park, Jan-Michael Santos, and Brad Wasson.

9. REFERENCES

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2005.

[2] M. Afergan, A. LaMeyer, and J. Wein. Experience with some principles for building an internet-scale reliable system. In *Proceedings of the 2nd USENIX Workshop on Real, Large Distributed Systems, WORLDS, San Francisco, CA, USA*, December 2005.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. F. andJ.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2003.

[4] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2003.

[5] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA*, June 2003.

[6] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang. Network imprecision: A new consistency metric for scalable monitoring. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA*, December 2008.

[7] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2003.

[8] Nagios. `http://www.nagios.org/`, 2010.

[9] E. Nygren, R. Sitaraman, and J. Sun. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3), July 2010.

[10] T. Repantis, X. Gu, and V. Kalogeraki. QoS-aware shared component composition for distributed stream processing systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(7):968–982, July 2009.

[11] A. Sherman, P. Lisiecki, A. Berkheimer, and J. Wein. ACMS: The Akamai configuration management system. In *Proceedings of the 2nd USENIX Symposium*

on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, May 2005.

[12] System Center Operations Manager (SCOM). `http://www.microsoft.com/systemcenter/en/us/operations-manager.aspx`, 2010.

[13] Visualizing Global Web Performance with Akamai. `http://www.akamai.com/html/technology/visualizing_akamai.html`, 2010.

[14] K. Wu, P. Yu, B. Gedik, K. Hildrum, C. Aggarwal, E. Bouillet, W. Fan, D. George, X. Gu, G. Luo, and H. Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Proceedings of the 33rd Very Large Databases Conference (VLDB), Vienna, Austria*, September 2007.

[15] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of the 2004 ACM SIGCOMM International Conference on Data Communication, Portland, OR, USA*, August 2004.