Distributed, Application-level Monitoring for Heterogeneous Clouds using Stream Processing

Michael Smit, Bradley Simmons, Marin Litoiu

York University
Toronto, Ontario, Canada

Abstract

As utility computing is widely deployed, organizations and researchers are turning to the next generation of cloud systems: federating public clouds, integrating private and public clouds, and merging resources at all levels (IaaS, PaaS, SaaS). Adaptive systems can help address the challenge of managing this heterogeneous collection of resources. While services and libraries exist for basic management tasks that enable implementing decisions made by the manager, monitoring is an open challenge. We define a set of requirements for aggregating monitoring data from a heterogeneous collections of resources, sufficient to support adaptive systems. We present and implement an architecture using stream processing to provide near-realtime, cross-boundary, distributed, scalable, fault-tolerant monitoring. A case study illustrates the value of collecting and aggregating metrics from disparate sources. A set of experiments shows the feasibility of our prototype with regard to latency, overhead, and cost effectiveness.

Keywords: cloud computing, monitoring, utility computing, distributed, Monitoring-as-a-Service

1. Introduction

There is broad consensus among those in enterprise and academia that the *cloud* [1, 2, 3, 4], at least to some degree, already exists. However, like all things, the cloud is constantly evolving. It is our belief, which we share with others, that today's idea of the cloud is rapidly morphing beyond its current, walled-garden model. In fact, there is broad interest in creating a cloud-of-clouds where multiple cloud providers can be accessed seamlessly, which we and others call the *multi-cloud*. Bernstein calls it the intercloud [5]; Buyya has described a vision for utility-oriented cloud federation [6]. Previous research from our research group explored a broker [7] and a metadata service [8] designed to support deploying applications that span multiple providers, and moving resources from one provider to another, autonomically.

A second extension to today's understanding of a cloud, that is being explored by the Smart Applications on Virtual Infrastructure (SAVI) national research project in Canada [9], proposes an architecture where virtualized resources exist close to end users (the *smart edge*), offering low-latency on-demand utility computing to any applications serving nearby users. Applications can move between the smart edge and traditional data centers (the *core*). Adaptive algorithms will manage this two-tier infrastructure, overseeing allocation of resources and the migration of applications automatically.

Hybrid clouds use public cloud resources to extend the capacity of a private cloud (when done temporarily, this is called *cloud bursting*). Open-source and commercial software to enable private and hybrid clouds are increasing in popularity, and open research problems include migrating application workloads from the private cloud to a public cloud.

The levels of automation and adaptation required to realize these federated, heterogeneous, hybrid systems require substantial monitoring information from both the system resources and applications involved. However, the heterogeneous nature of these hybrid systems-of-systems that span providers and networks makes monitoring difficult. The monitoring solutions that are currently widely deployed were designed to function at the cluster or provider level, and tend to be tightly coupled with a variety of metrics formats and vocabularies. While we continue to use these solutions

for the actual sensing of system-level metrics, the focus of this paper is on the collection, aggregation, and distribution of metrics from both the system level and the application level, to the application level.

We envision next-generation cloud systems exposing a monitoring-as-a-service (MaaS) interface, where regardless of the underlying provider or software, monitoring data from both the system and application layers is exposed in a consistent format, accessible as a web service. This service should support both the adaptation required to manage systems and the needs of users who use it to monitor cloud resources and applications. It should be capable of monitoring large numbers of resources and distributing the monitoring information to large numbers of clients, and scale easily as resources/clients are added and removed. It can rely on existing monitoring technology to obtain information at the resource level, but must move beyond that level to process, aggregate, unify, and distribute the monitoring information to subscribers.

This paper describes an architecture and a proof-of-concept implementation for meeting this need, called MISURE (Monitoring Infrastructure using Streams on an Ultra-scalable, near-Realtime Engine[1]). The architecture builds the monitoring infrastructure on a stream processing framework, such as S4 (Yahoo) or Storm (Twitter). These frameworks emulate the scalable, distributing processing of MapReduce but are intended to work on streaming data instead of stored data. Metrics are published by applications or acquired from resources using existing technology (Ganglia, SNMP, log files, Amazon CloudWatch). Monitoring data is pushed from resources to the monitoring system, where a series of stream processors converts incoming monitoring data to a unified format, producing streams of monitoring data. These streams can be processed by stream subscribers. We define three core stream subscribers that a) store monitoring data for later retrieval, b) push the monitoring data to remote services (replacing the repeated-polling approach that is common), and c) collate monitoring data from multiple sources to higher-level metrics. The architecture also specifies that end users can run their own stream subscribers using a platform-as-a-service (PaaS) [10] model. The entire system is scalable, and we specify a feedback loop that automatically deploys additional resources in response to increased demand. We test the throughput and latency of our proof-of-concept implementation, and evaluate the overhead. Using a distributed stream processor gives us a distributed, scalable, fault-tolerant, extensible system.

We define the requirements for MISURE more systematically in Section 2. Existing monitoring approaches in practice and in literature, as well as the details of a stream processing system, are described in Section 3. We present the conceptual architecture of MISURE (§4) and then describe our implementation (§5). A cost-monitoring case study (§6) demonstrates the value of such an approach. A set of experiments showing feasibility are shown in Section 7 before we conclude the paper (§8).

2. Requirements of Next-Generation Monitoring

Based on the next-generation cloud models described in the introduction, on existing monitoring solutions for infrastructure-as-a-service (IaaS)[10], and on the needs of adaptive management, we have established a set of requirements for a future cloud monitoring service.

Cross-boundary: A recurring theme of future cloud models is the collection of resources across multiple spheres of control; that is, the physical infrastructure is controlled by multiple different (sometimes competing) entities. For example, hybrid private-public clouds require monitoring both in the private data center and from the public cloud provider (the former is controlled by the user interested in monitoring data, but the latter is not). The core-edge model envisions edges in various geographic locations controlled by multiple providers, exposing similar interfaces but without guarantees of the underlying implementation (none of the resources are controlled by the user interested in monitoring data). The intercloud spans multiple public cloud providers, each of which controls their own infrastructure, and each of which offers different mechanisms of control of virtual infrastructure. In short, the monitoring service must accept monitoring data from multiple heterogeneous sources.

Scalable: The monitoring service should be capable of monitoring a small collection of resources, but scale to monitoring millions of metrics from thousands of servers and applications in multiple data centers or cloud providers, and provide monitoring data to thousands of end users or applications. This suggests horizontal scaling (adding nodes rather than increasing the power of individual machines), which suggests a distributed architecture.

[1] In Italian, the word *misure* means "measures".

Application-level monitoring: Operations staff are experts at monitoring machines and even virtual machines; existing monitoring solutions even in cloud services are predicated on this model. The vision of future clouds is that application developers will not know – or care – where their application is running, and machine-level monitoring will be less useful. The monitoring service should a) accept custom metrics from applications, and b) provide a means for aggregating raw metrics into more processed metrics at a higher-level of abstraction.

Fault tolerant: Receiving data from federated, distributed systems requires the monitoring system be resilient in the face of lost connections and missing data. Assuming a distributed architecture, the failure of any node must be recoverable without loss of monitoring data.

Near real-time: Future clouds require automated management due to their increased complexity, and automated management requires current and accurate monitoring data. The precise definition of "near real-time" varies depending on the use case. The ability of an aggregating monitoring service to meet this requirement depends on the timeliness of metrics received from third-party monitoring systems being aggregated. Similarly, the monitoring system can be run on the public cloud, which introduces more delays outside of the control of the monitoring system. Though near real-time is the goal, the requirement may be better expressed as "does not introduce excessive delay" to the distribution of metrics, where excessive is anything over one second.

Backwards compatible: The monitoring service should provide some of the functionality of existing services to end users, and to acquire monitoring data from existing sensors and monitoring systems. Existing implementations and tools should be incorporated where possible.

Uniform data structure: This requirement is related to the cross-boundary requirement: because the service is aggregating monitoring data from multiple sources, monitoring data will be in a variety of formats. For usability, these diverse formats should be converted to a common representation.

Self-managing: The components of the monitoring service should at minimum be able to scale up and down in response to changing workloads. Functional changes – such as reducing the granularity or frequency of monitoring data or disabling resource-intensive elements – may also be employed.

Push notifications: Existing solutions (e.g. Amazon CloudWatch) require applications to periodically poll the service to check for new metrics. While the monitoring service will also support polling (in accordance with the backwards compatibility requirement), it should be capable of pushing new metrics to registered applications.

Cloud-ready: The monitoring service should be built from the ground up to run on virtualized resources in a IaaS-type environment. This includes properties like minimizing disk reads/writes, high fault-tolerance, and tolerance for high-latency network communications.

3. Background and Related Work

Frameworks for monitoring have been proposed by several authors. Most of them deal with general distributed system performance monitoring or with particular domains such as computer networks or service oriented architecture [11]. The following section will provide an overview of these approaches.

Ganglia [12] is an open source[2] monitoring system for high performance computing and grid environments. Within a cluster, a monitoring daemon is run on each node collecting both built-in and user-defined metrics[3] while a multicast listen/announce protocol is used to transmit the details of the cluster's state to all member nodes. Across multiple clusters, federated monitoring is accomplished through the creation of a tree of point-to-point TCP connections. Periodically, a meta daemon (i.e., a non leaf node in this tree) polls its set of child nodes (i.e., monitoring and/or meta daemons), parses and persists the acquired data to a round-robin database and then transmits this aggregated data to clients. Ganglia is one of the originating sources of generated metrics that we federate in our approach. It is important to note that multicasting is typically not permitted on public IaaS providers; without this mechanism, the ability of Ganglia to seamlessly add and remove monitored resources is constrained, and the distribution and scaling of Ganglia reporting must be specified in configuration files.

[2] http://ganglia.sourceforge.net/

[3] These are published using a command-line program or a client side library.

Nagios[4] is an open source[5] tool for distributed systems monitoring. The service runs as a daemon and employs an architecture through which both system and host state is monitored by scripts/executables referred to as plugins. Anyone can write a plugin; however, several (both official and others) already exist and are available for download. State monitoring can be both daemon-directed, through a process referred to as an active check, or may occur in response to an external executable, through a process referred to as a passive check. Upon the observation of a state change (i.e., for a host or service) Nagios executes event handlers. One standard use for event handlers is performing management actions (e.g., restarting a service, logging to a database, etc.). Nagios is designed to support large-scale distributed monitoring and offers various approaches in support of this goal. However, the realization of certain distributed monitoring solutions is said to be challenging. Nagios could be used as an originating source of metrics for our monitoring service.

Apache Flume[6] is an open-source Apache incubator project designed to collect and store large amounts of log data to the Hadoop Distributed File System (HDFS). The MISURE system is also capable of performing this task, but the fundamental difference is stream-based processing deals with the flows of information, where each component performs an action on the stream. One processing task is to store streams to disk (we use HBase, which is built on HDFS), but we also have the ability to perform many other tasks: pushing metrics to end users, running client code to make decisions based on the run-time flow of metrics, etc.

The work in [13] describes a distributed monitoring service, implemented in Java and JINI (with WS-* bindings), called MonaLISA (Monitoring Agents in A Large Integrated Services Architecture). An agent in MonaLISA represents a service (i.e., that can be used by other services or clients) that is discoverable, self-describing and able to collaborate and cooperate with other services in performing various monitoring tasks. Collected data is stored, per service, in a local relational database. The Data Collection Engine directs MonaLISA's function. Clients may request both real-time and historical data through use of various filtering mechanisms (e.g., predicates, Agent Filter).

Clayman et al. [14, 15] describe the Lattice monitoring framework, designed to be a base framework on top of which monitoring systems may be built. Though in agreement with most of the requirements we specify here, their focus is more on the actual probes for sensing low-level metrics (e.g. CPU utilization probe). We are more interested in the collection, aggregation, and distribution of application- and system-level metrics from third-party probes.

The work in [16] introduces an architecture for and implementation of a private cloud monitoring system. The architecture is quite high-level and is composed of three layers: an Infrastructure layer, an Integration layer and a View layer. The implementation is modular in design and consists of several components that are mostly focused on the integration layer of the architecture. Currently, it is compatible with Eucalyptus (as a IaaS implementation); however, it is mentioned that it could be extended to work with alternative IaaS implementations in the future. It appears to rely quite heavily on Nagios for its monitoring functionality.

In [17], Kanstre and Savola define a set of requirements for a distributed monitoring framework and a reference architecture that satisfies those requirements. The requirements include scalability, correctness, security, adaptation and intrusiveness. The architecture is a conceptual layered architecture and there is no reported realization of it. An implementation of a distributed network monitoring framework was proposed in [18]. The authors showed how a three tier layered framework can be used for monitoring computer networks in geographically distributed locations. Compared with the above two approaches, our approach is better-suited to federated systems of clouds, though we share some common requirements such as scalability.

A cloud monitoring framework was proposed by Sun et al. [19]. The authors use a conceptual Service Oriented Architecture and focus mostly on message interchange among entities and on the integration of the framework with the existing system management processes. There is no implementation or evaluation of the architecture. We take the next conceptual step, recognizing the importance of scalability and intercloud monitoring on a loosely-coupled publish-subscribe architecture. We also provide an implementation and evaluation.

Lahmadi et al. [20] present a benchmark effort for defining metrics for evaluating a performance management framework. Their metrics include overhead, delay and scalability in the context of networks and services. While we use those metrics to evaluate our architecture, we focus on cloud environment and on a pub/sub architecture.

[4]More precisely, Nagios® Core™.

[5]http://nagios.org , http://nagios.sourceforge.net/docs/nagioscore/3/en/toc.html

[6]https://cwiki.apache.org/FLUME/

Balis et al. [21] described Gemini2, a monitor for grids built on a complex event processing (CEP) system called Esper (stream processing is in the same space as complex event processors). They suggest CEP is well-suited for monitoring as it enables access to streams of data in real time. They propose what amounts to substantial shift in how monitoring information is consumed: the end user writes and submits an SQL-like query requesting information, and the system deploys sensors to acquire this monitoring information which is then pushed to the end user. Support for existing monitoring services or even conventional monitoring paradigms is not included.

Amazon CloudWatch[7] is MaaS for Amazon Web Services resources (EC2 compute instances, storage volumes, etc.). Custom metrics can also be submitted. Metrics can be acquired using a command-line tool, a web service API, and a web-based GUI. Basic monitoring (5-minute polling frequency of 7 metrics) of EC2 instances is included in the price of the instance; detailed monitoring (1-minute polling frequency) is available for a monthly fee ($3.50). Custom metrics can be used for $0.50 per month. As one of the few MaaS solutions publicly available (Rackspace Monitoring is still in private beta), it is a key comparator for our infrastructure.

3.1. Stream Processing and Storm

Stream processing is in the family of complex event processing [22]; as an abstract concept, it refers to the generation, manipulation, aggregation, splitting, and transformation of data organized in a long sequence of records. S4[8], a Yahoo open-source project in the Apache Inbucator, and Storm[9], a Twitter[10] open-source project, are two examples of stream processing systems. We'll present stream processing by describing Storm in depth.

Storm is billed as a "distributed, scalable, reliable, and fault-tolerant stream processing system", and can be used for stream processing, continuous computation, and distributed RPC[11]. The key abstraction is a stream, which they define generally as an "unbounded sequence of tuples". The tuple consists of one or more fields; any tuple in a stream has values for the given fields. Two primitives, *spouts* and *bolts*, transform streams. A spout produces streams from existing data sources; a bolt takes an input stream and outputs a modified stream. Several bolts may be used in sequence to transform a stream. The spout and bolt abstractions are implemented by a set of tasks; each spout and bolt has a configurable number of tasks. (A spout and bolt can be scaled by increasing/decreasing the number of tasks implementing that primitive). When a spout task emits a tuple belonging to a stream, Storm chooses which bolt task should receive that tuple based on the *grouping* of the stream; it can be randomly distributed among the tasks, split up based on the field values (e.g., tuples with the same `id` field will always go to the same bolt), replicated to all tasks, or distributed to the nearest task (latency-wise).

A set of streams, spouts, and bolts is a *Storm topology*. Graphically, the topology is a directed graph where the vertices are spouts or bolts and the edges indicate which streams each belongs to, and their sequence. The topology is run on a *Storm cluster*. Streams fan-out from spouts, and fan-in to bolts: any number of spouts can have any number of connections to any number of bolts.

A Storm cluster has a master node and any number of worker nodes. The master runs a daemon called Nimbus with which worker nodes register. Topologies are submitted to the master, which distributes code and jobs to the worker nodes. A worker node may run any number of worker processes, each participating in various topologies.

Storm is parallel and distributed; there is no central router and no intermediate queue; spouts and bolts communicate directly based on the connection information in the topology. It is designed to scale horizontally. The elements of a cluster are all fail-fast and stateless; coordination is managed and state is maintained using a Zookeeper[12] cluster, so a loss of the master node or any worker node does not impact any running topology.

One of the key features of Storm is the effort to manage the complexity of distributed computation on realtime data entirely behind the scenes. This includes guaranteed message processing, aggressive resource management (garbage collecting defunct processes), fault detection and task reassignment after failure, efficient and scalable message transmission (using ZeroMQ[13]), streams that consist of any data (serialization occurs behind the scenes), and local de-

[7] http://aws.amazon.com/cloudwatch/

[8] http://incubator.apache.org/s4/

[9] https://github.com/nathanmarz/storm

[10] It was originally created by Nathan Marz of Backtype, which was acquired by Twitter.

[11] http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html

[12] http://zookeeper.apache.org/

[13] ZeroMQ is a socket library for concurrent message passing in various N-N communication patterns; see http://www.zeromq.org/.

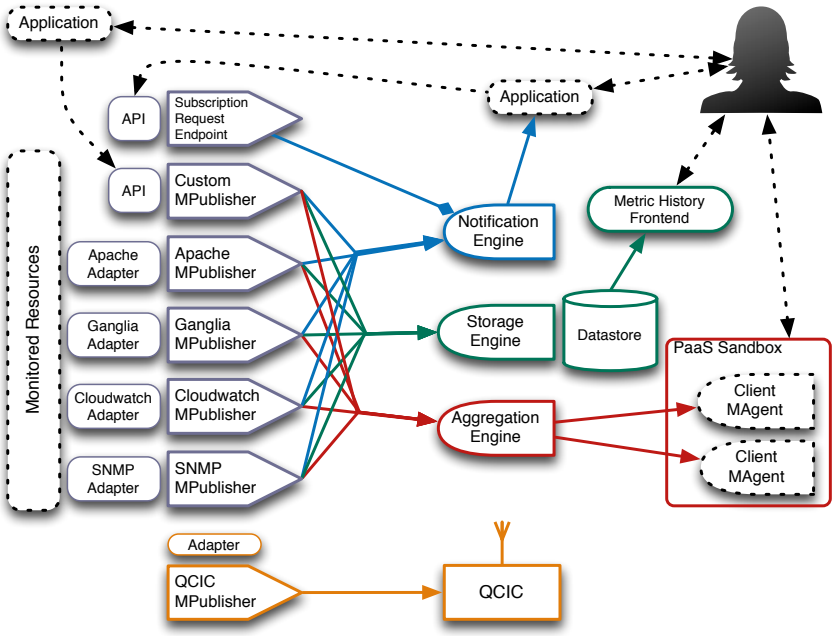


Figure 1: The MaaS overall design.

velopment environments for debugging. Storm allows spouts or bolts to be implemented using most programming languages.

4. An Architecture for Monitoring-as-a-Service

We designed an architecture for MaaS called MISURE to meet the requirements introduced in (§2) of a next-generation cloud monitoring infrastructure that could be deployed to offer MaaS to applications running on these platforms. We started from the basic publisher–subscriber pattern, where a set of publishers produce metrics without knowledge of the consumers, and a set of subscribers identify metrics to consume without knowledge of how they are produced. This loosely-coupled infrastructure is mediated by a broker; in this infrastructure, the functions of a broker are distributed among the various Engines, which are responsible for accepting published streams of metrics and requests to access streams of metrics. We use streams of metrics (*m-streams*) as the data elements to be produced and consumed, emanating from the producers, being transformed by the brokers, and terminating at a subscriber. An overview of the architecture is shown in Figure 1. Client connections and resources are shown using dashed lines. The following sections describe the key components of the architecture in more detail.

4.1. Streams

m-streams are *virtual streams* connecting a source to a destination; they may be implemented using several segments, they may be split or aggregated. Data that enters in one stream may leave in another, or in many. Regardless of implementation, conceptually there is a virtual stream that traverses the infrastructure end-to-end. Publishers are the first point where the metric enters the infrastructure, and subscribers are the first point the metric reaches when it leaves the infrastructure. (The subscriber is an end user or more likely a software component they provide). The key value of the infrastructure is in the editing, splitting, replications, aggregations, and other transformations made to the incoming publication streams to ensure the subscriber receives the data they request. A data stream is a useful abstraction and implementation for monitoring information. Streams connect a source and a sink uni-directionally, and often describe sequential information (e.g. video data). A stream can be multiplexed and de-multiplexed, which is

useful for scaling. There is no need to write a stream to disk; it can be generated, transmitted, and acted upon without being persisted.

At a practical level, an m-stream consists of well-specified metrics records; a record includes at least a timestamp, the source of the information, and the metric name and value (the arrows in Figure 1) A *c-stream*, instead of transmitting metrics, transmits specifically formatted control messages (lines terminating in a diamond in Figure 1). We also allow for *r-streams*, which is raw data received from a monitored resource that must be transformed into an m-stream.

4.2. Publishing Streams

The publishers have two roles, metrics acquisition and metrics conversion. Metrics are acquired from *originating sources* (lower-level existing components, applications, or services) as r-streams converted into m-streams. The original sources can be local machine monitors (like log files or data acquired using SNMP), local monitoring aggregation services (like Ganglia), or provider-specific monitoring aggregation services (like Amazon CloudWatch). The publisher can also monitor a machine, an operating system, or an application directly, but the goal is to re-use existing monitoring technology. The transformation from raw data (the r-stream) to an m-stream can often occur within a single component, but in some cases a series of components may be required (not shown; the design pattern of stream processing suggests multiple single-operation components is a preferable approach to distributed processing).

Metrics are converted to an m-stream and passed to the broker for promulgation. Sample **MPublishers** (Metrics Publishers, to distinguish from the general term in the pattern) are shown in Figure 1, along with the adapters that instrument or access the originating source. Though only one logical component is shown for each publisher component, there is no limit on the number of publishers, and the metric acquisition and conversion tasks for any given set of metrics can be split among multiple publishers. For example, multiple CloudWatch publishers can acquire monitoring data for different instances.

There are three special sources of streams included in the infrastructure. The **Custom MPublisher** (CMP) exposes a RESTful API to which applications can submit metrics; these metrics are converted to m-streams. The CMP is scaled using simple replication behind a hierarchical load balancer. The application is responsible for not sending duplicate messages; because the same CMP may not receive metrics from the application, there is no state or filtering imposed by the CMP.

The **Subscription Request Endpoint** (SRE) offers a RESTful API through which applications can subscribe to particular types of metrics by specifying the type of metric and/or the source of the metric, as well as a URI. The URI must accept JSON-formatted data submitted via POST requests. Unsubscribe requests may also be sent, as well as specially formatted control instructions to enable scaling (§4.3) The subscription information is passed to the Notification Engine for action along a c-stream. The Subscription Request Endpoint scales the same way as the CMP.

The **QCIC MPublisher**[14] serves as a meta-monitor, monitoring the components of the monitoring infrastructure. This m-stream is not available to general subscribers; it is consumed only by the QCIC component.

4.3. Subscribing to Streams

Setting aside internal pub-subs, the infrastructure provides three ways for virtual streams of data to transit the last mile (of the monitoring infrastructure) and reach the end user: notifications, querying stored metrics, and a PaaS model where clients add their own components directly to the infrastructure.

The push-based **Notification Engine** receives subscription requests (see §4.2) specifying an endpoint to notify for a type and source of metrics. All distinct streams of metrics (both raw and processed, see §4.4) are replicated to the Engine, which compares incoming metrics to the stored set of subscriptions. A set of notification agents transmit the metrics to the remote endpoints. The Notification Engine is scaled by splitting the incoming streams based on the source of the metric and launching multiple instances; the stream processing engine is designed to excel at this type of operation. The Subscription request stream is split the same way; although this stream is not high-volume, the subscriptions must be routed to the Engine instance that handles the specified source. For example, if five instances of the Notification Engine are launched, each will be responsible for approximately 1/5th of the resources being

[14] QCIC stands for Quality Control & Intervention Component, and references the Latin phrase *"Quis custodiet ipsos custodes"*, translated "Who will watch the watchers?". QCIC is used throughout to refer to the components responsible for monitoring and adapting the monitoring infrastructure itself.

monitored, with incoming tuples split among five streams based on the source of the metric. Subscription requests are based on the source of the metric, and will be split the same way.

An additional enhancement to this scaling strategy is required, because the Engine instance stores subscriptions that may become the responsibility of other instances if the topology is changed on the fly. For example, adding a sixth Notification Engine and splitting up the streams accordingly wouldn't work, because the subscription requests would be cached at the original five engines, and some of those sources would now be handled by the sixth engine. Therefore, we specify a loopback connection; when an Engine instance receives notice the topology is changing, it empties its cache of subscription requests into a stream that is passed to the Subscription Request Endpoint, causing the requests to be handled as if they were new and routed appropriately.

The **Storage Engine** implements a store-and-query pattern; incoming metrics are written to a datastore which can be queried through a web frontend or RESTful API (functionally similar to Amazon CloudWatch). While arguably the stream ends at the datastore, for consistency we consider the virtual stream to terminate at the end user, abstracting it as a query on a data stream. The most-recently stored metrics are cached in memory, preserving the requirement to limit IO delays. The datastore can be housed on a bare-metal machine, or high-IO IaaS instance. The datastore can be pruned of old entries periodically. The storage engine scaling relies on the scalability of the underlying DBMS. The Engine itself scales by adding additional instances. The web / service frontend scales using multiple instances and hierarchical load balancers, all reading from the same datastore.

The **PaaS Sandbox** allows end users to run their own stream processing code, receiving raw streams of metrics and performing custom processing. The components provided by the end user are called Client Monitoring Agents (MAgents). An API is provided for identifying streams to subscribe to and receiving metrics. An example use case is a simple adaptivity solution: an MAgent can monitor streams of metrics and add or remove resources as appropriate, all from within the PaaS Sandbox. The end user can specify whether their MAgent should be scaled horizontally based on the volume of metrics; otherwise, metrics it cannot process will be discarded.

4.4. Brokering Streams

There is no one component responsible for brokering the streams. However, the **stream processing system** plays a key role in establishing the links among components, managing queues, transmitting data, etc. A key function is splitting streams in different ways for scaling; sometimes the incoming metrics can be split evenly among several streams (e.g. Storage Engine), while other times a more precise split is required (e.g. based on the source of the metric, as for the Notification Engine). Stream multiplexing (combining multiple streams into one stream) is done using helper components (not shown) that subscribe to multiple data streams and produce a single data stream. Scaling the stream processing system relies on the underlying implementation.

The **Aggregation Engine** processes raw metrics to produce processed metrics at a higher level of abstraction. For example, application metrics and server-level metrics can be aggregated to correlate application performance with server utilization. Though shown in the diagram as connected to the PaaS Sandbox, as a common use case of these processed metrics is the client MAgents, processed metrics are also available to the Notification and Storage engines.

The **QCIC** receives monitoring data from the QCIC MPublisher (QMP) that includes only the components involved in this architecture. All of the components can be scaled using the scaling approach defined for each, both at deployment time and adaptively at runtime. The QCIC uses the stream processing system to send control signals to the various components involved in the architecture. The QCIC is also responsible for implementing dynamic monitoring policies that alter the frequency, quality, or other attributes of monitoring systems in response to the environment (for example, [23]).

5. Implementation

We implemented a MISURE prototype on top of the Storm (0.60) stream processing system (§3.1)[15]. We used Java; the total implementation is approximately 2,500 lines of code. There is a natural mapping between publishers and Storm spouts and between m-streams and Storm streams that forms the basis of our implementation.

[15] A demo video is available at http://www.ceraslabs.com/projects/monitoring-next-generation-cloud-systems or by contacting the authors; the live demo is not always available due to the EC2 charges).

Our implementation is currently used as the basis for a cost monitoring project (one use case is described in §6), and has been used to monitor resources in a project where a private cloud is overlaid on public resources [24].

5.1. Streams

The Storm stream abstraction is an unbounded sequence of tuples; in our implementation, each tuple includes fields for a timestamp, the name of the metric, the value of the metric, and the source of the metric. Metric names are in the form `class.name[.subname]`. The types of class include any valid application name (`mysql`, `apache`, etc.), server, os, etc. The name, and optional subname, describe the metric from that item (`mysql.num_queries`, `estore.cart.abandoned_count`, etc.). The fields in c-streams vary depending on the stream. The data in the streams is serialized behind-the-scenes by Storm, using a configurable serialization method or Java Serialization.

5.2. Publishing Streams

Each publisher is implemented as a spout (or in some cases a spout+bolt combination) that produces m-streams, following a similar design pattern. An agent (or a set of agents, as appropriate) acquires logs or metrics from the originating source. Where possible, these are converted to an m-stream immediately and emitted. Where not possible – for example, an Apache log file does not have an obvious conversion to this format – the log entries are sent along an r-stream to a helper bolt. Based on the control messages, an m-stream is created (in our example, aggregating raw log records to requests per time interval metrics).

The sample **MPublishers** shown in the architecture diagram (Fig. 1) have been implemented as we expect they will be generally useful. The Apache MPublisher provides adapters that are installed on the Apache server; these adapters replace the Apache log file with a FIFO socket so Apache writes logs to the adapters in memory instead of to disk. The adapter forwards the raw log text to the spout, which passes the logs to a helper bolt for conversion to an m-stream (as described above). The Ganglia MPublisher provides an adapter that is installed inside a cluster to monitor broadcast metrics (if Ganglia is configured to log to a central system, the adapter must be installed on that system). The adapter forwards the raw metrics to the spout implementation which produces an m-stream. The Cloudwatch MPublisher is provided with a configuration document that lists the instances to monitor and the metrics to gather, as well as whether to operate at 1-minute or 5-minute frequency (and AWS credentials). Multiple adapters are launched to asynchronously connect to CloudWatch and acquire metrics periodically; the spout implementation produces an m-stream by aggregating input from the adapters. The SNMP MPublisher is provided with a configuration document specifying the ip address/port/protocol combinations which should be queried, and the oids it should poll at each location. The configuration also specifies a mapping from oids to text names for use when producing the m-stream. Asynchronous adapters are launched to connect to the SNMP hosts and acquire the metrics. Additional MPublishers can be added using the simple extensible, pluggable architecture; we describe such extensions in §6.

Recall that Storm does not use intermediate queues; a spout only sends a tuple when the bolt is ready. Error handling is required to detect when the Spout is acquiring metrics faster than the bolts can process them.

The **Custom MPublisher** accepts JSON-formatted requests specifying the name, value, source, and timestamp of the metric. The submitted data is sanity-checked and then converted into a m-stream. The **Subscription Request Endpoint** API accepts JSON-formatted requests specifying the source and the name of the metrics that should be sent to the provided endpoint, and produces a c-stream consumed by the Notification Engine. The **QCIC MPublisher** is based on the Ganglia MPublisher; the Storm cluster deployer automatically starts Ganglia monitoring on all resources involved in the cluster.

5.3. Subscribing to Streams

The subscription services provided by the infrastructure are implemented by bolts, or sets of bolts. Bolts receive m-streams, and emit m-streams, so can be composed. (Bolts may also receive r-streams and emit m-streams, e.g. §6.)

The **Notification Engine** bolt maintains a list of metrics sources and names that end users have subscribed to via the Subscription Request Endpoint, and compares incoming metrics to that list (a hash table based on the metric name offers more efficient lookups). An asynchronous thread pool is used to send notifications to the requested endpoints. To demonstrate the end-to-end function of the virtual stream, we also implemented the client side of push notifications. The client-side application is a web portal for visualizing metrics (Figure 2). Because of web browser

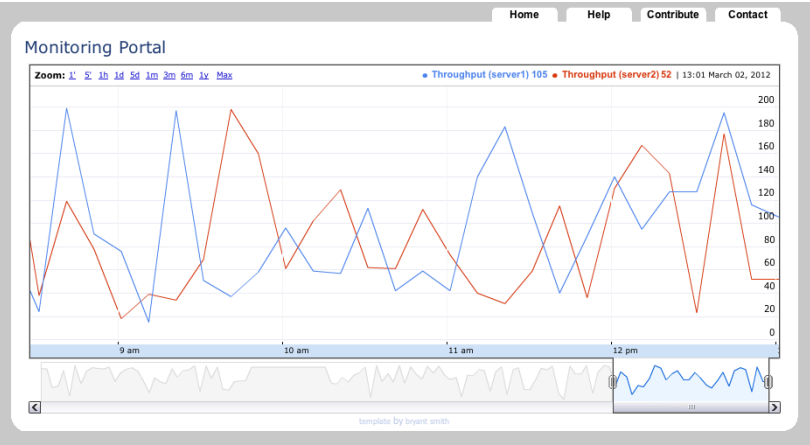


Figure 2: Our web front-end receiving push notifications from the monitoring service (screenshot).

security restrictions[16], we use an intermediate PHP web application which implements and exposes the RESTful web service specified by the Notification Engine. When the Notification Engine pushes a metric to that web service, it stores it (using memcached to store state across multiple sessions) and pushes it to the Javascript application using Ajax[17]. The Google Visualization API[18] is used to visualize the data.

The **Storage Engine** is a bolt that listens to incoming m-streams and writes metrics to the Open Time Series Database (OpenTSDB)[19], which may be running on a separate machine. OpenTSDB uses HBase to store and promulgate metrics in a distributed, scalable, and standards-driven way. End users can access metrics using a web front end (Figure 3), a RESTful API, or a command-line tool. This provides a polling-based subscription method that offers functionality similar to existing tools. OpenTSDB storage scales using the HBase scaling features; the retrieval service scales using standard load balancing techniques.

The **PaaS Sandbox** is not entirely implemented as specified; while bolts can be authored and added to the Storm topology, and a set of APIs exists to create Client MAgents, the security is not in place to actually allow execution of client code. Storm in its current state would not offer the security needed, and must be modified to support this feature.

5.4. Brokering Streams

The stream brokering functions are handled almost entirely by Storm. In particular, sophisticated stream splitting is offered out-of-the-box to distribute streams among the various bolt tasks; the bolts can be scaled by adding or removing tasks. Scaling is also provided by the Storm implementation. Although we have an adaptation system we intended to use for the QCIC component, run-time modification of Storm topologies is in the roadmap for Storm but is not yet implemented. The challenge of aggregating metrics to produce processed metrics at a higher level of abstraction varies depending on the domain; one example is illustrated in a case study (§6).

6. Case Study: Calculating Cost

One higher-level piece of information difficult to measure in a multi-cloud environment is the total cost of infrastructure services for a given deployment. Consider Amazon EC2, for example. Each instance is billed for the

[16] Even with Javascript, a web page cannot open a socket and listen on it.

[17] Our implementation uses the long-poll design pattern to emulate push notifications, which until the complete implementation of Web Sockets is the *de facto* standard.

[18] https://developers.google.com/chart/interactive/docs/gadgetgallery

[19] http://opentsdb.net/

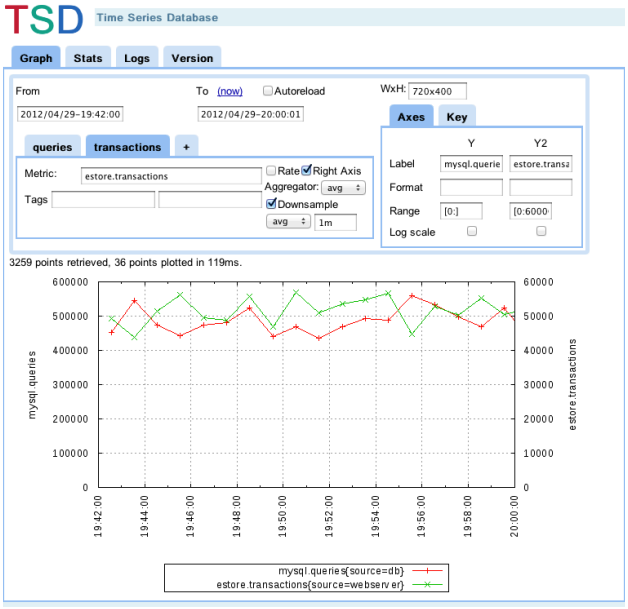


Figure 3: The OpenTSDB front-end retrieving metrics stored by the monitoring infrastructure (screenshot).

allocation of compute resources, but also incurs consumption based pricing for outgoing bandwidth on public networks, data transfer within a datacenter, storage (when using Elastic Block Storage (EBS), the default), and block storage read/writes. Additional optional services may add additional charges, for example detailed CloudWatch monitoring, EBS optimization (provisioned levels of IO operations), additional public IP addresses, and automatic load balancing. There are also a variety of pricing models (on-demand instances are the standard, but reserved instances involve an upfront fee and a reduced hourly rate, while spot instances have variable pricing as users bid on spare capacity). The Amazon Web Services API offers the ability to get an estimate of the current monthly bill, and the billing site offers the ability to download spreadsheets listing services and charges after they occur, but no other information is made available. The API also does not provide information about some of the services that are billed; for example, there is no API to retrieve a count of disk operations or of how much outbound traffic is public ($.12/GB) or private ($.00-$.01/GB).

This is only one provider. In a multiple-provider situation, each provider has their own prices and their own billing practices and cost models. In the private cloud, depending on the software used to deploy, resource metering and billing may not even be possible. For example, at the time of publication, the popular open-source private cloud manager OpenStack has an efficient metering system in active design and development[20].

The complexity of measuring the cost of infrastructure services prevents several interesting use cases: for example, monitoring resources to identify under-utilized infrastructure that can be released; monitoring cost of infrastructure versus income generated by a deployed application to detect cost-of-service attacks [25] (or to measure efficiency in general); or determining how much a deployed application is costing, and what it might cost with a different mix of providers. This case study examines a use case similarly unsupported: we use both Amazon EC2 and a local Openstack install to implement and test research ideas and applications. We are interested in knowing what we are currently spending per-hour on Amazon, and how much we are saving when we run experiments on Openstack instead (applying the Amazon cost models to our local resources, and reporting that as saved money).

To realize this use case, we added several components and streams to our MISURE architecture as shown in dashed/red lines in Figure 4 (some components from Figure 1 have been removed for readability). We added two new MPublishers (implemented as Spouts). The CloudyMetrics spout reads metadata about instances and price from a cloud metadata service [8], which includes pricing for public cloud providers and for any local resources used (for

[20] http://wiki.openstack.org/EfficientMetering

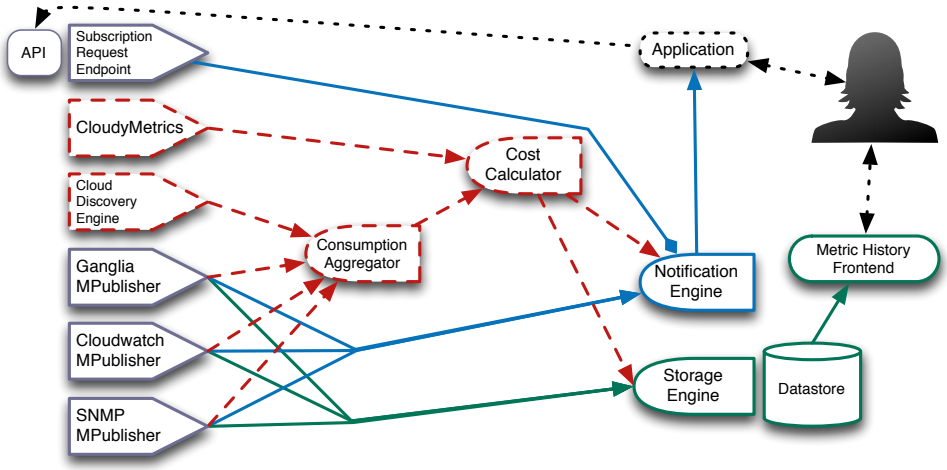


Figure 4: The cost monitoring components (dashed/red) and streams, integrated into the MISURE architecture.

our use case, the same pricing as Amazon EC2), and produces an r-stream with the cost data (the cost information is used to generate other information, rather than being stored and handled by the monitoring infrastructure, and so is never converted to an m-stream). The Cloud Discovery Engine MPublisher connects with all known cloud providers (based on the presence of properties files with authorization credentials) using JClouds[21] as an abstraction layer. Some provider-specific code is required (for example, for Amazon EBS). It produces an r-stream that includes a data structure representing all resources it could discover: compute, storage, bandwidth, etc.

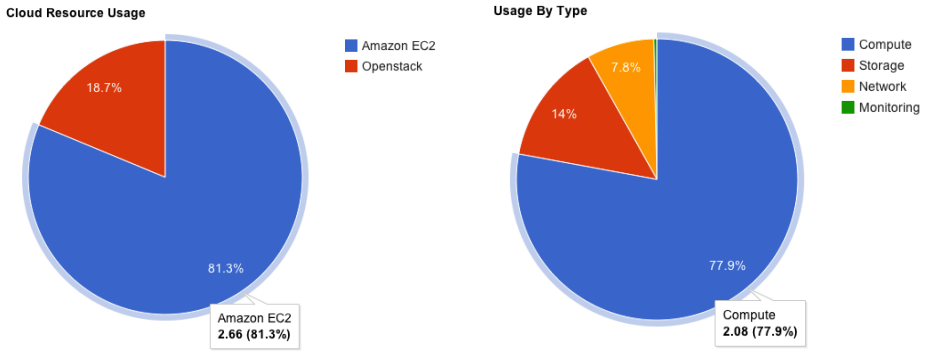
A new bolt, the Consumption Aggregator, accepts the r-stream from the Cloud Discovery Engine, and adds all available monitoring information in the form of m-streams: Ganglia, CloudWatch, SNMP, etc. These inputs are in the unified metrics tuple format, so the originating MPublisher is not important, and in fact any other MPublishers could also provide m-streams. We deployed two custom monitoring probes and integrated them to the Ganglia monitoring system: one to measure disk I/O, and one to measure outgoing data transfer to public networks. This data is included in the m-stream. The role of the Consumption Aggregator is to decorate the data structure received from the Discovery Engine with consumption information based on incoming metrics: network resources are augmented with data transfer amounts, storage with I/O reads, etc. A second new bolt, the Cost Calculator, accepts this decorated data structure, as well as the cost information r-stream from the CloudyMetrics MPublisher, and further decorates the data structure with current rate information (cost per hour, or per GB, or per disk read, etc.). The Cost Calculator then produces an m-stream in the unified format, providing for cost information every resource, as well as various aggregated totals: total per provider; total per resource type; total for an instance taking into account compute, storage, and network for that instance; and so on. This m-stream passed through the rest of the MISURE system (notification engine, storage engine, PaaS MAgents, etc.) and can be subscribed to just like any other metric.

We deployed this extended implementation to Amazon EC2, configured to monitor a hybrid cloud (our private cloud and Amazon EC2 for the public cloud). We amended our monitoring portal (§5) to add new graphs showing the hourly cost breakdown by provider (satisfying our use case) and by resource type (for additional information), shown in Figure 5. We see we spend approximately $2.66/hour on Amazon EC2, but save about $.61/hour by using our local Openstack resources (annual savings of > $5000).

7. Experimental Evaluation

In addition to the case study we use to illustrate the value of aggregating monitoring data from multiple sources and processing it in a stream-based fashion, we also conducted a series of experiments as an early exploration of the

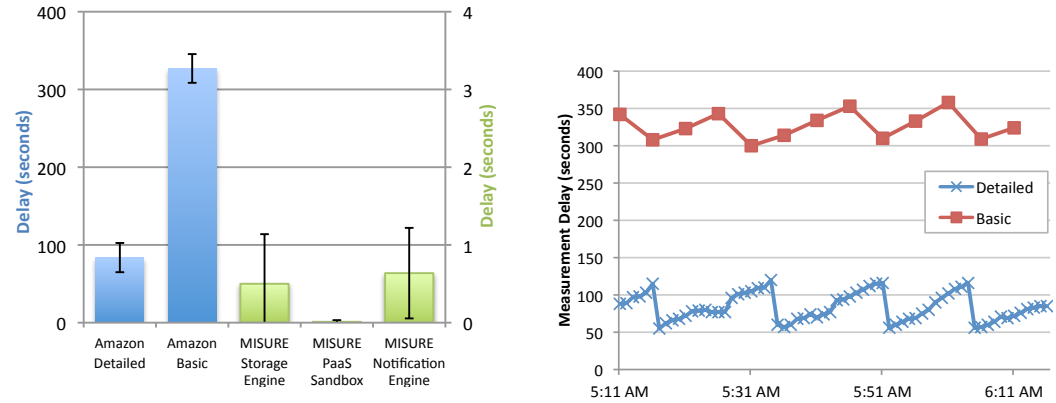
[21] http://www.jclouds.org



(a) Current EC2 costs, and what we save by using local resources.

(b) Breakdown by type for Amazon EC2

Figure 5: Graphs generated by our cost monitoring case study (screenshot).



(a) Average measurement latency; the three right-most columns use the right axis. n = 100

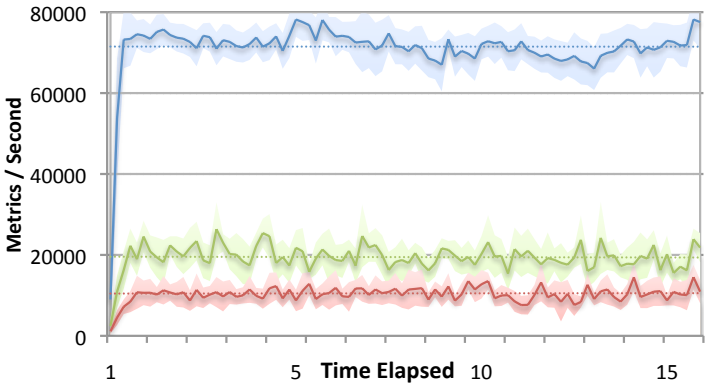
(b) CloudWatch measurement latency

Figure 6: Measuring latency between when a metric is recorded and when it reaches the end user.

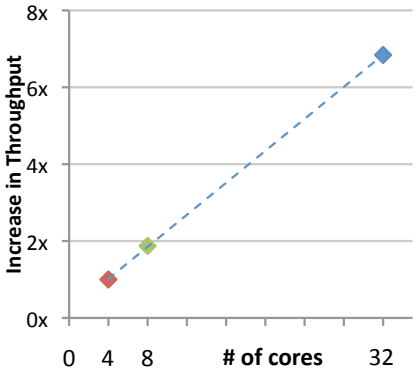
feasibility of our prototype implementation regarding the non-functional requirements of near-realtime, scalability, and fault-tolerance.

Experimental Storm Cluster. We created a storm cluster on Amazon EC2, using m1.large (dual core) instances (all verified to run Intel Xeon 5507 processors) across multiple availability zones. We deployed a single master node that included the work assignment daemon Nimbus, the Zookeeper instance, a Ganglia metadata daemon to collect metrics from the cluster, and the Storm web-based status monitor to track deployed topologies. This master node did not register any substantial utilization during the experiment as the infrastructure is highly decentralized. A variable number of worker nodes were added depending on the experiment; Storm allows worker nodes to dynamically join and leave the Storm cluster. To this we deployed the implemented monitoring infrastructure storm topology, modified as described in the following experiments. We deployed an OpenTSDB interface using an HBase installation comprised of several high-IO instances.

Functional Requirements. In functional testing, our implementation gathered and successfully stored metrics from a bare-metal mysql server using SNMP, metrics from Amazon EC2 instances, and custom application-level metrics artificially generated by a testbed application. (Cross-boundary, Application-level, and Backwards compatible requirements). Our web front-end received push notifications (Figure 2), though the Google chart used can handle only a limited number of metrics at a reduced update frequency. An HTML5 canvas app would address these shortcomings. The OpenTSDB interfaces (web, command-line, and API) allow rapid querying with a high cache-hit rate for recent



(a) Average throughput for 4 (red), 8 (green), and 32 (blue) cores across 30 samples, with standard deviation (shaded).



(b) Increased throughput from 4 to 8 and 32 cores, averaged over 450 minutes per infrastructure size.

Figure 7: Metrics per second processed by the monitoring infrastructure running at full capacity, both for a 15-minute time window (a) and overall (b).

metrics (tens of milliseconds), and querying of historic metrics took 1-2 seconds even after storing 100 million metrics.

Near real-time. The exact definition of near-realtime depends on the application; for monitoring, we set a target of 1 second or less for the complete transit of our infrastructure. To test this, we measured *measurement latency*, the delay between when a measurement is made and when that metric arrives at the end user's application. With the federation role of our infrastructure, the measurement latency depends on the originating source which varies widely; thus, we focused on how much latency the infrastructure adds from the time the metric becomes available.

To eliminate the delay of an originating source, we used a simple application to randomly generate metrics at 1 second intervals, with a spout to insert these into the monitoring infrastructure. We then used the various retrieval methods to retrieve metrics: a command line tool to access the Storage Engine and then immediately run the `date` command to see the time of the query, a modified version of our notification client that would print the time a metric was received at the Visualization API, and a custom bolt running in the PaaS sandbox that measured the latency[22]. For comparison, we ran similar tests using the command line tool to retrieve Amazon CloudWatch metrics for a single instance. The PaaS sandbox ran on the Amazon EC2 cloud; all the other tests sent queries to our testbed from another network.

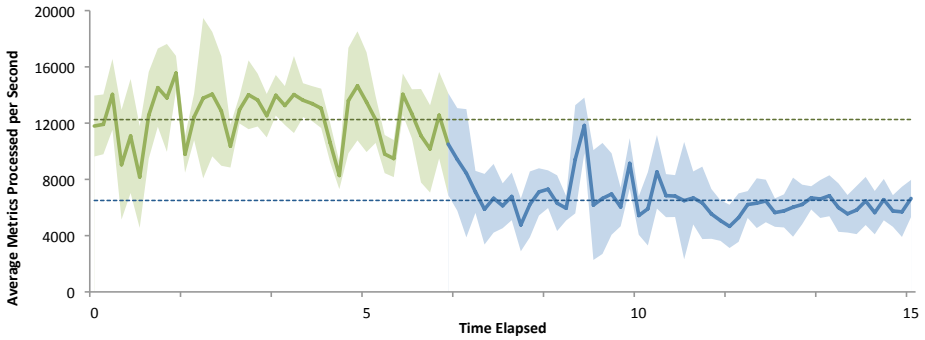
The results are shown in Fig. 6a; the error bars are the standard deviation over 100 samples. The CloudWatch delay (first two bars) uses the left axis, and the monitoring infrastructure uses the right axis, a difference of two orders of magnitude[23]. PaaS Sandbox had the lowest measurement latency, but all the monitoring infrastructure subscription methods were sub-second.

We noted an ongoing increase in latency when using CloudWatch, followed by an abrupt adjustment downward (Fig. 6b). For the detailed monitor, this abrupt adjustment resulted in two metrics appearing at the same time. The period between abrupt adjustments is not constant. The result is end users may not realize how current the incoming metrics are.

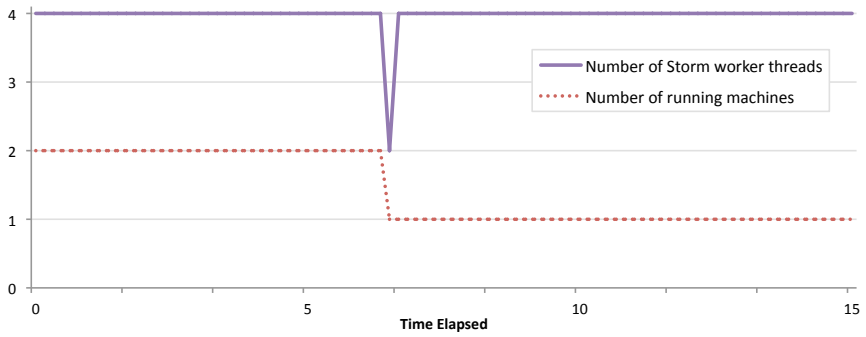
Cost-effective, Scalable, and Fault Tolerant. A key factor for monitoring systems is overhead. The overhead on the resources themselves depends on the originating source, and is outside the scope of this infrastructure. However, the infrastructure itself requires resources; we require that the infrastructure not be cost prohibitive. A series of throughput experiments were designed to measure throughput and in so doing demonstrate scalability, cost-effectiveness, and fault tolerance. Note that these features are provided in part by the existing Storm infrastructure;

[22]It should be noted that measuring the exact instant a new metric appeared is not possible; there is delay in retrieving and recording the current time, the two timestamps are from different clocks (note we did synchronize clocks using NTP), etc.

[23]The substantial delay in basic monitoring is due in part to CloudWatch using a metrics timestamp from the start of the period. The metrics timestamped 5:10 are an average of the values from 5:10-5:15, and are reported some time between 5:15 and 5:16.



(a) Average throughput before and after a crashed instance, over 10 samples, with standard deviation (shaded).



(b) Storm threads recovered from a crashed instance.

Figure 8: Demonstrating fault tolerance, showing a failed instance.

however, a Storm topology must be designed to maintain scalability and fault tolerance. These experiments ensure that the monitoring implementation does not impair the ability of Storm to scale or recover from errors.

To measure throughput, we ran a series of experiments with varying numbers of cores and machines: 4 cores on two machines (with four Storm workers), 8 cores on four machines (with eight Storm workers), and 32 cores on sixteen machines (with thirty-two Storm workers). We used the same application as in the latency experiment to generate random metrics, except it generated metrics as quickly as possible; the number of workers dedicated to generating metrics was manually tuned to completely utilize all resources without overloading the system. Each metric was stored by the Storage Engine, processed by a bolt in the PaaS Sandbox, and pushed by the Notification Engine, though only to one client. Each task ran in parallel depending on the number of workers; up to 10 workers were used for a single task, each processing one-tenth of the metrics. Data was recorded for a total of 450 minutes (almost two billion metrics for the 32-core infrastructure).

The average throughput over all samples (n = 30, 6 experiment runs with 5 samples per time interval) is shown in Figure 7a for 4, 8, and 32 cores (solid line). The 32-core monitoring infrastructure averaged 257 million metrics per hour. The standard deviation is shown in the shaded area, and a trendline is fitted to each set of data points (dashed line)[24].

Our scalability is provided primarily by Storm[25], so to evaluate the monitoring infrastructure we must ensure that our bolts and spouts as implemented scale horizontally. The scaling policies for each component were described in Section 4. As a first test to ensure we had maintained scalability, we examined the throughput experiment. All cores registered 100% utilization during the experiments. In Figure 7b, we plotted the improvement factor when moving from 4 cores to 8 or 32. Increasing the number of cores by a factor of 8 increased throughput by a factor of 6.85, which

[24]Though only 15 minutes of data are shown, the experiments all ran for 90 minutes

[25]http://storm-project.net/about/scalable.html

suggests a scaling efficiency of 86%. We believe improving the performance of the prototype bolts would improve the scaling efficiency.

Fault-tolerance was demonstrated using a similar experimental configuration without the Storage Engine, with four cores (and four worker threads) on two machines workers and two applications randomly generating metrics. Partway through the experiment, one machine was crashed (without warning, by launching a process to consume all of the memory until it became unresponsive). Though there is a noticeable drop in throughput (Figure 8a; standard deviations in shaded area, n=10), the affected tasks were re-allocated to the remaining machine (Figure 8b), and processing continued.

To assess cost effectiveness, we compare this approach to using Amazon CloudWatch. The full capacity of Cloudwatch is not known so a direct throughput comparison is not possible; however, we can do a cost analysis. Detailed monitoring is priced at $3.50 per month, and includes 7 metrics recorded every minute and stored for two weeks (approximately 300,000 measurements per month). This puts the per-measurement cost at 1.15 cents per 1,000 measurements (the basic monitoring is free; we assume the cost is factored into the price of the instances). Our 8-core cluster required five m1.large instances and could process over 70 million measurements per hour, worth $818.89/hr in revenue at Amazon monitoring prices. The total cost for the 5 on-demand m1.large instances in our cluster is $1.65/hour (which will increase as more metrics are stored; CloudWatch retains metrics for two weeks). While the comparison is not rigorous – we have yet to test thousands of clients retrieving metrics simultaneously – these results are promising.

8. Conclusion

The next generation of cloud-oriented systems will require a novel approach to monitoring that crosses boundaries, federating millions of metrics from heterogeneous sources, while supporting existing monitoring services and monitoring paradigms. The increased importance of adaptive systems for managing complex systems-of-cloud-systems highlights the importance of accurate, timely, uniform, cost-effective monitoring data. We have defined a set of requirements for monitoring-as-a-service that meets the needs of these services.

We have described an architecture and an implementation called MISURE that meets these requirements. Built on a stream processing framework called Storm, metrics are managed as streams, which provide near-real time, in-memory access for both publishing and subscribing. The pluggable, extensible architecture allows for metrics from existing sensors to be published as streams. These streams are made available to subscribers by push notifications, by pull polling, and by a pluggable architecture allowing the subscriber to provide their own component to manage streams. Architecturally, streams can be aggregated, and the entire monitoring infrastructure can be managed adaptively. A case study illustrated the need for such a monitoring system, and an implementation targeted at the case study demonstrated ability to produce aggregated metrics from a set of low-level metrics. Experimental validation showed the capacity for millions of metrics each hour at low cost, with low measurement delay, and performance that increases linearly as more resources are added.

As Storm continues to evolve, more of the MISURE architecture will be implemented, particularly the adaptive management piece adding and removing components and workers at runtime. A promising item on the roadmap is a state spout, which will send updates only when the state of an object (as determined by several metrics) has changed. This could be a useful construct for a monitoring system, where metrics need only be transferred if they are different. We intend to continue development of this monitoring infrastructure, and currently intend to use it to monitor hybrid clouds and two-tiered cloud architectures.

Acknowledgements

This research was supported financially by IBM Centres for Advanced Studies (CAS), the Natural Science and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network, Ontario Centre of Excellence (OCE) and Amazon Web Services (AWS); these agencies had no input into the design or execution of this research. Thanks to the members of the CERAS Labs for feedback on an early version of this infrastructure, particularly to Mark Shtern.

References

[1] B. Hayes, Cloud computing, Commun. ACM. 51 (2008) 9–11.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the Clouds: A Berkeley View of Cloud Computing, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[3] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, F. Galan, The reservoir model and architecture for open federated cloud computing, IBM Journal of Research and Development 53 (2009) 4:1 –4:11.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Comp. Syst. 25 (2009) 599–616.

[5] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, M. Morrow, Blueprint for the intercloud - protocols and formats for cloud computing interoperability, in: Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services, IEEE Computer Society, Washington, DC, USA, 2009, pp. 328–336.

[6] R. Buyya, R. Ranjan, R. N. Calheiros, Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services, in: ICA3PP (1), pp. 13–31.

[7] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, S. Mankovski, Introducing STRATOS: A cloud broker service, in: IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 891 –898.

[8] M. Smit, P. Pawluk, B. Simmons, M. Litoiu, A web service for cloud metadata, in: IEEE Congress on Services, IEEE Computer Society, Los Alamitos, CA, USA, 2012, pp. 24–31.

[9] SAVI, Strategic network for smart applications on virtual infrastructure (savi), http://www.savinetwork.ca/, 2011. Last access 02/01/2012.

[10] P. Mell, T. Grance, The nist definition of cloud computing, National Institute of Standards and Technology 53 (2009) 50.

[11] P. Hershey, Soa monitoring for enterprise computing systems, in: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International, p. 443.

[12] M. L. Massie, B. N. Chun, D. E. Culler, The ganglia distributed monitoring system: design, implementation, and experience, Parallel Computing 30 (2004) 817 – 840.

[13] H. Newman, I. Legrand, P. Galvez, R. Voicu, C. Cirstoiu, Monalisa: A distributed monitoring service architecture, Arxiv preprint cs/0306096 (2003).

[14] S. Clayman, A. Galis, C. Chapman, G. Toffetti, L. Rodero-Merino, L. M. Vaquero, K. Nagin, B. Rochwerger, Monitoring Service Clouds in the Future Internet, in: Towards the Future Internet - Emerging Trends from European Research, IOS Press, 2010.

[15] S. Clayman, A. Galis, L. Mamatas, Monitoring virtual networks with lattice, in: Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP, pp. 239 –246.

[16] S. De Chaves, R. Uriarte, C. Westphall, Toward an architecture for monitoring private clouds, Communications Magazine, IEEE 49 (2011) 130 –137.

[17] T. Kanstrén, R. Savola, Definition of core requirements and a reference architecture for a dependable, secure and adaptive distributed monitoring framework, in: Dependability (DEPEND), 2010 Third International Conference on, pp. 154 –163.

[18] S. Hongjie, F. Binxing, Z. Hongli, A distributed architecture for network performance measurement and evaluation system, in: Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on, pp. 471 – 475.

[19] Y. Sun, Z. Xiao, D. Bao, J. Zhao, An architecture model of management and monitoring on cloud services resources, in: Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on, volume 3, pp. V3–207 –V3–211.

[20] A. Lahmadi, L. Andrey, O. Festor, Performance of network and service monitoring frameworks, in: Integrated Network Management, 2009. IM '09. IFIP/IEEE International Symposium on, pp. 815 –820.

[21] B. Balis, B. Kowalewski, M. Bubak, Real-time grid monitoring based on complex event processing, Future Generation Computer Systems 27 (2011) 1103 – 1112.

[22] D. Luckham, The power of events: an introduction to complex event processing in distributed enterprise systems, Addison-Wesley, 2002.

[23] N. Villegas, H. Muller, G. Tamura, Optimizing run-time soa governance through context-driven slas and dynamic monitoring, in: Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the, pp. 1 –10.

[24] M. Shtern, B. Simmons, M. Smit, M. Litoiu, An architecture for overlaying private clouds on public providers, in: 8th International Conference on Network and Service Management, CNSM 2012, Las Vegas, USA, pp. 371–377.

[25] J. Idziorek, M. Tannian, Exploiting cloud utility models for profit and ruin, in: Cloud Computing (CLOUD), 2011 IEEE International Conference on, pp. 33 –40.