Semantic Based Data Collection for Large Scale Cloud Systems

Jonathan Stuart Ward
School of Computer Science
University of St Andrews
jonathan.stuart.ward@st-andrews.ac.uk

Adam Barker
School of Computer Science
University of St Andrews
adam.barker@st-andrews.ac.uk

ABSTRACT

Current tools for monitoring cloud systems are designed for physical servers and are not intended to handle rapid elasticity or dynamic behaviour while operating at scale. Though current monitoring tools can be applied to small cloud systems, the volume of data and computational overhead associated with their operation render them unsuitable for large scale cloud deployments. The metrics obtained by current solutions also lack a machine readable structure, limiting the ability of both software and humans to interpret the data. As cloud adoption continues, the scale and complexity of cloud systems will present significant challenges to current tools. This paper proposes a scalable distributed data collection system which forms the basis of a cloud monitoring system. Utilising technologies from the semantic web, our architecture generates a machine readable overview of a cloud system without the need for an additional dedicated monitoring system. We present an exemplar implementation of our architecture written using the Python programming language and perform an evaluation demonstrating its ability to provide scalable data collection services fit for cloud computing.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Systems

General Terms

Cloud Computing, Semantic Computing

Keywords

Cloud Computing, Semantic Computing, Large Scale Systems

DIDC'12, June 19, 2012, Delft, The Netherlands.

1. INTRODUCTION

Prior to the advent of cloud computing, large scale systems were only available to sizeable institutions or through volunteer computing. Since the popularisation of cloud computing, previously unobtainable scalability has become available to all organisations and users via a utility model. Smaller institutions and individuals can now deploy large scale systems by leveraging an Infrastructure as a Service (IaaS) cloud for significantly less investment than required for physical servers.

A distributed system operating within a cloud expresses a number of unique properties not found in systems based on physical servers. Through cloud computing, a system can change entirely in composition and function within a single hour invalidating many past techniques for managing change. A technique which has migrated from physical systems to cloud computing with little modification is monitoring which remains pivotal but unchanged.

Systems monitoring is a critical function within any distributed system, allowing for the detection of failure, misconfiguration and poor performance. Monitoring is especially critical for large distributed systems where events and behaviours may not be as easily detected. Monitoring is conventionally provided by an additional set of separate servers which operate independently from the systems which they monitor. Most current monitoring solutions require a set of dedicated servers which either actively poll members of a system for data, or has clients push data to it. This requires a number of monitoring servers proportional to the number of hosts and services being monitored and the analytics being performed [8]. When collecting and analysing a large number of metrics, this makes monitoring a large scale system extremely data intensive and extremely costly in terms of the required hardware and bandwidth. The data requirements become a significant concern given the metered data services of many cloud providers.

Despite the difficulties in monitoring large scale systems [10], the same software tools used for monitoring physical systems are now extended to handle cloud systems. Monitoring a system composed of a fixed number of constantly operational physical computers is a computationally intensive task. Monitoring a system hosted on a cloud which is prone to rapid changes in scale and composition presents a significant challenge. As the size and complexity of systems deployed on a cloud increases the volume of monitoring data and the rate of change in the system will put increased strain on centralised client-server based monitoring tools. For cloud adoption to continue uninhibited, new de-

centralised distributed monitoring tools are required to replace the current tools which are unfit to meet the demands of the emerging generation of large scale cloud systems.

The difficulties of using software to monitor large dynamic systems are compounded by the difficulties human administrators face in attempting to understand the composition of cloud systems. A server running on physical hardware has a unique identity based upon several factors: location, specification, manufacturer, configuration, domain name and so forth. This allows for a localised view of a system when a function or activity can be easily understood to occur at a specific location. When virtual machines (VMs) are deployed in a small number, humans can still understand the purposes and interactions of individual VMs however when deploying hundreds or thousands of VM instances it becomes difficult to differentiate the purposes and activities of two otherwise identical VMs. This phenomena reduces the value of a low level view of a system. Furthermore, as VMs within a single deployment are usually instantiated from a very small number of images, the configuration and resources of these hosts are identical. This further depreciates the value of an individual VM and instead encourages a more holistic, system wide view.

To obtain a holistic view requires the availability of information from the entire system. Obtaining of host and network data is traditionally fulfilled by a monitoring service, however the data obtained by current solutions lack any machine readable structure limiting the value of the data. While this limitation could be potentially overcome through complex analytics, the lack of a highly scalable monitoring solution able to handle rapid change significantly limits the value of monitoring. To meet the needs of the emerging class of large scale elastic cloud systems we propose a new class of distributed data collection system. Based upon technologies from the semantic web, our proposed architecture is a scalable peer-to-peer system which allows a large system to be self describing without requiring a complex external monitoring system.

2. MONITORING REQUIREMENTS

Examination of the challenges presented by large scale cloud systems allows us to derive the set of requirements which a large scale cloud monitoring service must meet in order to function optimally. Our architecture aims to fulfil each of these requirements.

Decentralised: systems which collect data at a single location scale poorly, act as a bottleneck and as single point of failure [2] and require greater bandwidth. This is unacceptable for large scale cloud systems, instead data collection must be a distributed process which allows for superior scalability.

Location Aware: cloud computing has a considerable variety of costs - both in terms of capital expenditure and in terms of performance. Data transfer between different VMs hosted in different regions can incur significant financial costs, especially when dealing with big data. Monitoring data will eventually be sent outside of the cloud in order to be accessed by administrators. In systems hosted between multiple clouds there will be both inter and intra cloud communication. Both these cases can result in poor performance due to latency and cost due to metered data. Latency which presents a significant problem for application running on the cloud [4], including monitoring. When monitoring physical servers a host can be but a few hops away, cloud computing gives no such guarantees. This will adversely affect any monitoring system which polls according to a given schedule and otherwise produce delay. A location aware system can significantly outperform a system which is not location aware [6] and reduce the costs inherently associated with cloud computing. Hence a cloud monitoring system must be aware of the location of VMs and collect data in a manner which minimizes delay and the costs of moving data.

Fault Tolerant: failure is a significant issue in any distributed system, however it is especially noteworthy in cloud computing as all VMs are transient. VMs can be terminated by a user or by software and give no indication as to their expected availability. Current monitoring is performed based upon the idea that servers should be permanently available. As such current monitoring systems will report a failure and await the return of the failed server. A cloud monitoring system must be aware of failure, and of VM termination and account for it appropriately. Crucially such failure must not impede the operation of data collection and monitoring.

Autonomic: having to configure a live VM, even a trivial configuration, is a significant overhead when dealing with large numbers of VMs. Current monitoring systems require both a central server and the monitored VM to be configured before monitoring can begin. Existing monitoring solutions geared towards cloud computing attempt to solve this problem by using automated scripts to perform configuration. This however results in monitoring being configured with generic settings which may be non ideal and requires additional software which needs updated and maintained in addition to the monitoring software. This method may be appropriate for small scale systems where only simple metrics are required but this method is insufficient for larger systems. Software based on the now obsolete paradigm of static system deployment is unsuitable for large dynamic systems [11]. Monitoring large scale cloud systems requires an autonomic monitoring system, that post VM instantiation requires no human configuration.

Holistic: large scale cloud systems necessitate a macro level view pertaining to the relationships and interactions of components. Large scale systems express complex emergent behaviour which cannot be easily described by independent metrics obtained from throughout the system. Due to the presence of identical VMs in vast quantities the value of individual metrics from a single point in a cloud system is minimal. Monitoring data from a large scale cloud system needs to be considered within the greater context for it to be valuable. This requires contextual metadata for each datum collected by a monitoring system and a means to quantify the relationships and behaviours arising as a result of individual values.

```
"Server 1"      rdf:type        server:server
"Server 1"      server:runs     "DB Server"

"Server 2"      rdf:type        server:server
"Server 2"      server:runs     "Web Server"
"Server 2"      server:runs     server:JVM

"DB Server"     rdf:type        db:MySQL
"DB Server"     db:stores       "table A"
"DB Server"     db:stores       "table B"

"Web Server"    rdf:type        server:Apache
"Web Server"    web:serves      "Web App"

"Web App"       rdf:type        web:app
"Web App"       db:writes       "table B"

"job 1"         rdf:type        cron:job
"job 1"         db:sanitizes    "table A"
```

Figure 1: Two servers represented using the Terse RDF Triple Language (TURTLE)

Selective: Fundamental to monitoring is the collection of the noteworthy state of a host and related network segments, however with cloud VMs, much of the state is identical and represents little interest to the end user. The homogeneity of VMs instantiated from the same image results in few factors of differentiation. A cloud monitoring system need capture only the state of a VM which is unique to that VM and other factors which are constant in all VMs instantiated from the same image can be discarded.

In order to meet this set of cloud monitoring requirements we propose a new distributed monitoring system based upon semantically linked data. By doing so we provide a basis for the monitoring and management of large scale cloud systems which are becoming increasingly prevalent in business and academia. As the scale of cloud systems increases, the ability for humans to manage individual components is diminished. Human administrators can configure a finite number of components and cannot keep up with a rapid rate of change in system composition. This necessitates the need for automated configuration and autonomic adaptation and fault correction. This cannot be achieved easily without data providing a holistic view of the large scale system, the data which our architecture attempts to provide.

3. SYSTEM ARCHITECTURE

3.1 Semantic Data Collection

Unlike current monitoring solutions, our solution does not use a database or flat files to represent data. Instead our solution is based upon the Resource Description Format (RDF), a W3C recommendation which defines a method for modelling semantic metadata [17]. RDF allows statements based upon a subject-predicate-object expressions to describe a resource. The vocabulary used by RDF is extensible via additional schemas [3], allowing for comprehensive ontologies representing any problem domain to be generated. The key benefit of using RDF is that it produces a datastore of machine-readable information which can be accessed, analysed and interpreted without the need for significant configuration and interaction. Figure 1 shows a RDF representation of a group of servers using the Turtle language [18] which can be easily parsed by software. Figure 2 shows the same RDF data modelled visually in a human readable fashion using a relational diagram. The human and machine readable properties of RDF which makes it extremely valuable for monitoring purposes.

Semantic data contains explicit definitions of each datum and the relationships between data. Any software with access to the same RDF vocabulary that is used to define the data can formulate complex queries to establish how data is related and the importance of each value. This key benefit allows semantic data to be autonomously processed while traditional monitoring and data collection systems use relational databases which are not machine readable and limit their autonomic applications. In addition to being machine readable, semantic data also maps well to natural language. This allows for a semantic datastore to be parsed and returned to a user as a textual description of a system or to produce readable activity logs and allows for natural language queries of a system. It is therefore the ideal basis for a data collection system which serves to fulfil the previously defined requirements for cloud monitoring.

3.2 Components

Our architecture is based upon a series of nodes ordered within a peer to peer structure. A node is defined as a a cloud VM running the architecture's software. Figure 4 illustrates the interactions of each of the components present within a single node. Each node operates the following:

- **Triplestore:** typical monitoring and data collection services rely on centralised relational databases to store data obtained from a set of hosts. This limits scalability and results in a significant volume of data transfer. Instead of a relational database our architecture uses a series of Triplestores [12] as its storage system. A Triplestore is a datastore optimised for the storage and retrieval of triples, such as RDF statements. Each node operates its own triplestore populated with data collected regarding itself and possibly other nodes within the architecture.
- **Data Collection Daemon:** each node collects data based upon a schema provided in the VM's image. Data is collected both locally, and dependant upon a node's location within the architecture, from the triplestore of other nodes. The pertinent services and variables are monitored and upon a noteworthy change (with pertinence and noteworthiness being described via the schema) the value is written to the triplestore. Similarly, dependant upon its role in the architecture, a node may at a schema defined interval collect data from other nodes. Local data collection is facilitated through a series of Open Source libraries and structured according to a set of RDF vocabularies before being committed to the triplestore.
- **Query Interface:** each node's triplestore is accessible via a SPARQL Protocol and RDF Query Language (SPARQL) query interface. SPARQL [16] is an RDF query language which allows the unambiguous querying of semantic data. Other nodes and external agents

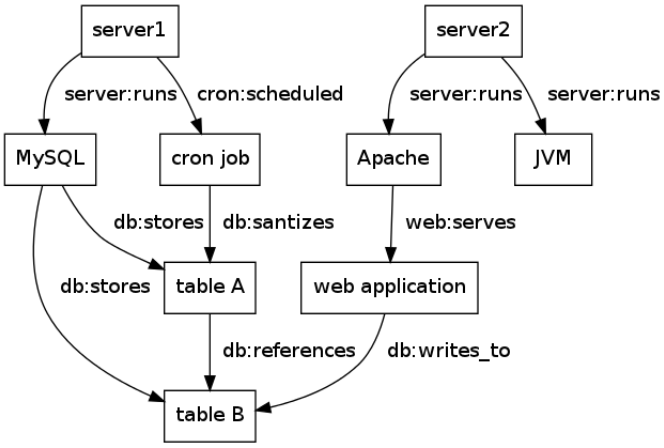

Figure 2: A more human readable diagram representing the same data and generated from the RDF in Figure 1

can access the architecture via the query interface to obtain the available data.

- **Maintenance:** each node runs a component which manages it's behaviour and location within the architecture and communicates with other nodes.

3.3 Operation

Prior to deployment of the architecture a VM image is created containing the necessary software its intended purpose and the node software. Deployment of the architecture begins upon deployment of a VM or set of virtual machines. The architecture is intended to operate over hundreds and greater numbers of VMs, while it will function with smaller numbers this is not its primary goal. For this description, consider several hundred VMs being instantiated. The first four of the VMs to be instantiated attempt to bootstrap themselves. The bootstrap service may be DNS, a web service, an online storage service or other resource. Upon failing to find an available bootstrap node offered by the bootstrap service, the first nodes will register themselves as bootstrap nodes with the chosen service. The nodes which have become a bootstrap node then form a bidirectional ring network between themselves.

3.3.1 Bootstrapping

Once the bootstrap nodes are available, the next VMs which are instantiated will connect to one of the nodes returned by the bootstrap service. Upon connecting to a bootstrap node the node will form a distributed maximal binary heap, whereby the initial bootstrap nodes serve as the root of the heap and all successive nodes, the children. The value used for ordering each heap is a integer value representing a prediction of longevity derived from an analysis of data collected at each node. Each node will periodically reevaluate its precoded availability based upon newly collected data. Upon a child deriving a greater prediction of availability than its parent a swap will be performed. Hence the node with the greatest prediction of uptime will ascend to the root of the heap, while nodes predicted to be available for less time will descent to the leaves. Upon a node replacing the root node it will become the new root and part of the ring and will assume the neighbours of the previous node. While all nodes instantiated to perform a set task are likely to have identical availability predictions this does not affect the architecture. It is envisioned that our architecture will operate over systems with frequent changes in composition due to VMs being instantiated and terminated and where there may be significant variation in predicted availability. While prediction of longevity is the default value used for ordering, additional or alternative factors can be used, including system load.

3.3.2 Balancing Heaps To Balance Load

When a node is bootstrapped it is sent to the smallest heap in an attempt to keep each heap balanced. A heap may become disproportionally smaller than other heaps due to a large number of VMs being terminated. In this case, following an agreement between ring nodes, the other heaps will reassign nodes to the smaller heap. Should the overall number of nodes exceed the number of nodes within each heap exceed the number of by a factor defined by the end user, the number of ring nodes will be doubled and new heaps created at each. The immediate child with the greatest longevity prediction of each ring node will be made a new ring node and the heaps will be balanced between each other. A similar process or regression will occur if the number of children are significantly less than the number of root nodes. This ensures that no single root node is burdened by a substantial heap and ensure that there is no unnecessary data transfer due to an unnecessarily large number of heaps.

3.3.3 Data Propagation

Each node collects data regarding itself based upon its predefined XML schema. Each node also assimilates the triplestore of each of its children. Hence, every node stores the data of every child of greater depth within its branch. The root of each heap therefore stores the cumulative data of the entire heap. The roots/ring nodes serve as the entry point to external agents. The same method for bootstrapping nodes is used to locate ring nodes in order to perform a query. A SPARQL query is created either by hand or more likely by automated tools and sent to a ring node. The ring node queries its triplestore and forwards the query to each of its neighbours with forward the query in turn. Each

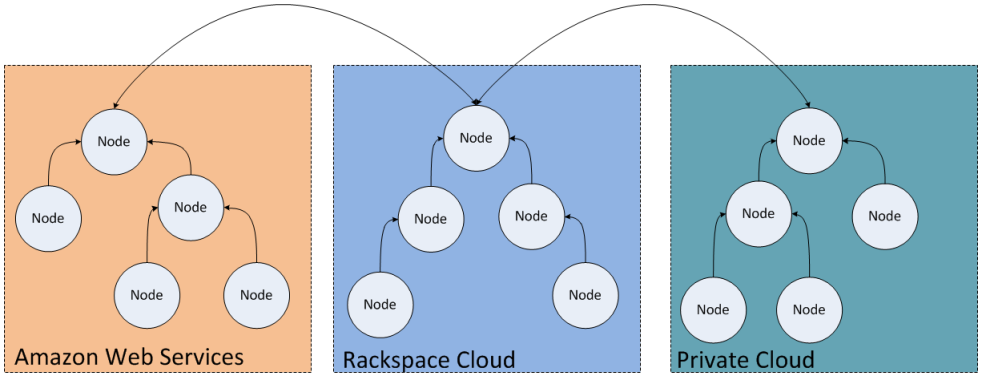

Figure 3: Multiple data collection heaps distributed across clouds for optimal performance and reduction in costs and overhead

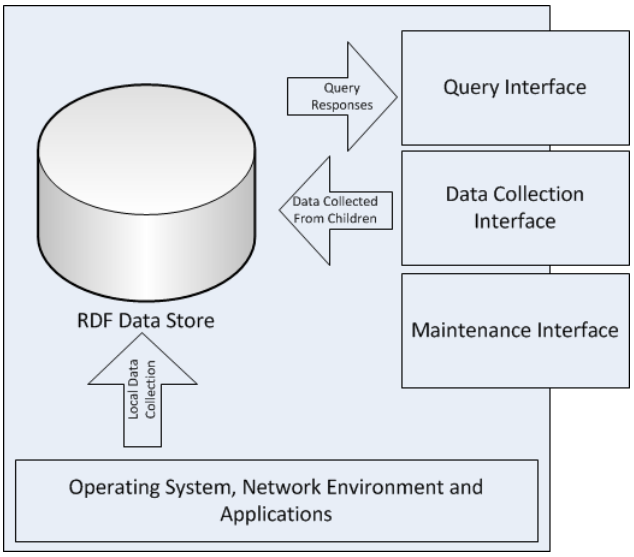

Figure 4: The components operating within a node

ring node returns the result of the query to the node issuing the query which in turn returns the combined results to the client.

3.3.4 *Failure*

Failure and the termination of VMs is inevitable in a large scale distributed system. By ordering components based upon a prediction of longevity, the impact of failure should be minimized by locating failure likely VMs to the leaves of each heap. Should however the predictor of longevity prove inaccurate, nodes which are higher in the heap may be terminated, rendering entire subtrees disjoint. In such a case the roots of the two new disjoint heaps will rejoin the heap from which they were disconnected at the point closest to their previous location. As each node stores the collective ontology for each of its children the amount of data propagation required upon reconnecting is minimal or even zero depending upon the next free location. Only the roots of the disjointed heap are aware of the failure, children within each of the heaps are agnostic to the failure and continue operating as expected.

Having entire subtrees disconnected from the system is

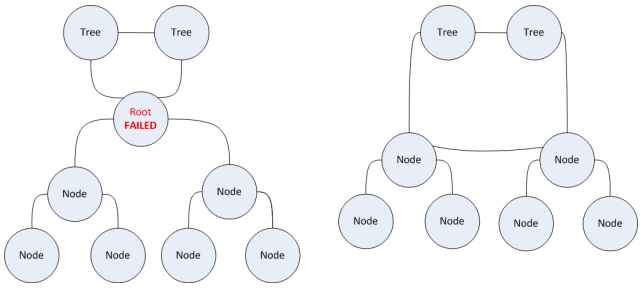

Figure 5: Policy in the event of a root node failure, large subtrees advance to the ring level

problematic and prevents coherent monitoring from occurring while the subtree remains disjoint. Should a node close to the root (again defined by the user) or the root itself fail in lieu of disconnecting large subtrees those disjoint subtrees can be elevated to the ring becoming two entirely new heaps and removing any need for rebuilding the tree. As shown in Figure 3 this maintains consistent trees when encountering failure. This can be set as the default behaviour for handling failure however overall performance and consistency is reduced when the ring frequently changes.

This architecture therefore collects an ontology representing a distributed system through a joint tree and ring structure. This allows a system to be self descriptive without the need for an external monitoring system.

4. USE CASES

Unlike a traditional monitoring system, our architecture simply facilitates data collection and at present does not serve to detect faults and alert administrators of changes in state. It could be modified to do so, however this is not the most interesting application of our architecture. Instead this architecture serves to provide a holistic view of a large system as a whole and ideally disregards values which are identical between VMs or otherwise irrelevant. In addition to serving as a platform for future research this high level viewpoint has number of use-cases which a conventional monitoring system cannot easily achieve either due to scale, complexity or limitations in data representation.

4.1 Autonomic Computing

Given the scale of many systems deployed on a cloud managing individual VMs is a difficult, tedious and time consuming process. This has given rise to significant interest in autonomic computing to reduce the management overhead associated with cloud computing. Of the four functional areas of autonomic computing as defined by IBM [5] our architecture provides the greatest benefit to self-optimisation. As stated by IBM, a precursor to autonomic optimisation is autonomic monitoring. A system must be able to collect data regarding itself in order for it to be able to perform any optimisation based upon the newly acquired knowledge [2]. Our architecture provides an ideal framework for autonomic data collection in large scale systems. Once a VM image has been created with all the necessary software, our architecture requires no further interaction and will continue to operate until there are no longer any active VMs. This allows our architecture to serves as a basis for a number of different autonomic computing functions based upon the machine readable data which it makes available.

Conventional monitoring tools capture data at the micro level and fail to capture the complex emergent properties which are only evident at the macro scale. Distributed systems seldom operate in a vacuum and events occurring in one part of the system often have direct implications on events in a separate location. While these events could be detected by examining multiple sets of metrics for different hosts, by instead querying a set of machine readable semantics an autonomic process can examine a distributed system with far greater ease. While the metrics may hold significance for a human, they hold no explicit meaning for an autonomic system and do not easily allow for automated decisions to be made. Semantically structured data however has explicitly defined significance and relationships to other data which allows the relevance and importance of the available data to be considered. This is ideal for self-optimising systems as it allows the direct affect of any optimisation to be examined and its effectiveness determined.

4.2 Identification of Redundancy and Volunteer Computing

Our architecture provides an ideal facility to identify unused resources. VMs and storage resources are often allocated for a set purpose but are not used to capacity in fulfilling that purpose. Similarly, users within an organisation may deploy VMs which replicate existing functionality which was otherwise not known of. Our architecture allows the identification of unused resources and the location of resources matching a specific requirement which reduces the likelihood of replicating existing resources. Should an organisation deploy our architecture it would be feasible to locate redundancy, replication of function and unused capacity and either terminate these resources or reuse these resources for a useful purpose. As volunteer computing systems are usually set to operate when there are free resources or the system is idle the available resources within a systems can by quickly identified using our architecture in order to quickly dispatch a set of volunteer jobs. A key bottleneck in volunteer computing is scheduling and job distribution [1] which limit how many jobs can be run and job throughput. With the availability of a holistic view of a system, volunteer computing jobs can be executed and terminated based upon events throughout the system. For example, should an application's frontend become idle, useful work could be scheduled on the back end as no work is expected. With our architecture, obtaining this information is far easier than with a non semantic system.

4.3 System Modelling

Unlike conventional monitoring systems which simply obtain metrics, our architecture is capable of representing a system semantically. The collected data can be visualised and be analysed in order to facilitate a post hoc investigation of the previously monitored system and the data can be modified to investigate potential optimisations and reconfigurations without requiring a live system. This has a number of other uses including aiding in postmortem diagnosis of failure, security breaches or other significant problems and can aid in the formal verification of system behaviour.

5. EXISTING WORK

There are a number of monitoring systems which are capable of collecting data from large scale systems. These can be broadly classified into systems which push data from its point of collection and those which pull data and into systems which are fully centralised and those which are more decentralised. Nagios represents a relatively centralised system which pulls data from each monitored component which Ganglia is more decentralised and pushes data from its point of collection.

5.1 Nagios

Nagios [9] is the de facto standard Open Source monitoring system. With an extensive community and large user base Nagios has an comprehensive plugin library affording it support for monitoring numerous applications, devices and configurations. Initially conceived as a centralised monitoring tool, current versions of Nagios are capable of some degree of distributed monitoring. Nagios supports two architectures for distributed monitoring:

- Offloading data collection and verification duties to a series of slave nodes to reduce the load on the centralised master. The slaves contain no configuration and simply execute the orders of the master. This allows for the instantiation of additional slaves to handle load. The slaves however do not perform complex functions such as data analysis or graphing which are performed by the master. Thus, this architecture suffers from a central bottleneck and point of failure which limits its scalability.

- An alternative architecture allows for monitoring functions to be divided between multiple monitoring servers. Each server is provided a portion of the infrastructure to monitor and performs all data collection, checks and analysis for that portion. A centralised front end then aggregates data from each of the monitoring servers.

While distributed, these architectures rely on additional servers to monitor a given infrastructure. For monitoring transient cloud VMs this serves as a significant overhead both in terms of cost and configuration. Owing to its heritage as a fully centralised monitoring solution Nagios also suffers from a series of limitations which reduce its viability as monitoring solution of large highly dynamic systems:

- Nagios follows a polling model and will poll monitored resources according to an automatically generated schedule. A polling schedule is generated to minimize the overall load that the monitoring server incurs and to ensure the balanced polling of each resource. The schedule will be dynamically altered based on failure, delay or other unexpected results. Rapid changes in scheduling can result in excessive work, increased delay between pollings and in the extreme case; inability for the monitoring server to keep pace with the schedule. When monitoring resources which are prone to sudden change, this presents a significant issue.

- Nagios is typically configured by an administrator defining a set of configuration files which describe the resources, servers and devices which are to be monitored. This process is static and limits the ability to monitor dynamic systems. There are multiple solutions to this problem provided by third party plugins and services operating in addition to Nagios. The most solution is to use a configuration management service, such as Puppet, to automatically generate and apply new configuration dynamically. This requires additional management, configuration and software which presents unnecessary overhead and complexity.

In addition to the potential architectural problems, Nagios uses either flat files or an SQL database as a storage medium. This provides users and applications with unlinked and relatively unstructured data which limits the inherent usefulness of the data. While many of the issues pertaining to monitoring large dynamic systems with Nagios could be overcome through plugins and significant modification its problems stem from its inception as a centralised solution. A fully decentralised architecture is necessary for the challenges presented by cloud computing which at present Nagios cannot provide.

5.2 Ganglia

Ganglia [7] [14] is a scalable system monitor intended for high performance systems. Unlike Nagios it is highly decentralised and was from the outset designed to monitor large scale systems. Each server being monitored by Ganglia runs a data collection daemon which announces changes in state to other members of a defined group. Each node thus maintains a representation of the entire group which is made accessible via an XML based query interface. Nodes within a group, order themselves within a hierarchy to propagate data and respond to queries. The data pertaining to each group is federated via a meta daemon. A web based front end then makes available the federated representation of the entire system. Ganglia requires only a minimum set of additional servers to collect data and present a web interface which most of the functionality occurring on the systems being monitored.

Originating from the domain of high performance computing, Ganglia primarily obtains low level metrics including load and resource usage. As such, it is not intended to capture higher level state pertaining to applications and system behaviour. The data which is captured is represented either using XDR or XML, which lacks the inherent structure and value of semantically linked data.

While Ganglia does offer an architecture which is well suited to disseminating and collecting data in a large scale system it was designed intended to do so within an HPC environment. High performance computing is typically done within one, or relatively few locations and has a dedicated high speed network. Cloud computing however operates extensively over multiple separate data centers, providers and connections. This heterogeneous network environment is not the preferred environment for Ganglia and requires significant separation and organisation of groups of monitored systems to ensure optimal operation.

Ganglia also fails to make any provision for highly dynamic behaviour. The HPC environments which Ganglia was designed to monitor typically do not exhibit dynamic behaviour unlike those involved with cloud computing. While capable of some dynamic behaviour within a group, Ganglia requires a significant amount of manual configuration to create a new group. This again makes Ganglia non ideal for monitoring highly dynamic systems.

6. EVALUATION

6.1 Prototype

This architecture is implemented in a prototype developed using Python 2.7. The prototype, a product of rapid application development, consists of four separate python programs:

The Node: responsible for managing location and communication within the architecture

Data Collection: using standard UNIX tools for collecting system data it populates a datafile which the node propagates

Bootstrap Service: a service which allows nodes to register themselves as a ring node and for clients and new nodes to locate ring nodes

Client: a simple command line client which can be used to query a node

6.2 Evaluation Methodology

To evaluate our architecture we provide an empirical comparison against Ganglia and Nagios to demonstrate that our architecture is better suited to cloud monitoring and will result in traffic and overhead and hence offer superior performance and reduced financial costs. Our evaluation is based on four premises:

1. Collected monitoring data will eventually be sent out with the cloud to an end user.

2. Monitoring systems will be configured with their default and most basic settings. This is typical of monitoring which is configured by automated scripting and represents the unlikelihood of complex configuration being performed to large numbers of VMs en mass.

3. VMs are instantiated from the same image.

4. Bandwidth is a metered resource (as is typical in most IaaS clouds).

Our evaluation was conducted using the St Andrews Cloud Computing Colaboratory Cloud [15], a private cloud based upon Eucalyptus [13]. Each monitoring system was deployed

over 25 VMs and one external physical server. 25 VMs provides a sufficient sample size to indicate traffic and network behaviors however is insufficient to produce large volumes of data, future evaluation will consider far greater numbers of VMs. The physical server serves to represent the host which eventually accumulates data from the VMs within the cloud. Conceptually this could be a monitoring server, storage server or a user's computer. Each configuration attempted to follow the de facto configuration that is most frequently found in real world use, in each case the configuration entailed:

Nagios: 25 VMs running the Nagios Remote Plugin Executor and one external server which polls NPRE on each VM.

Ganglia: 25 VMs running *gmond* to perform monitoring functions and the physical server running *gmetad* to obtain data from each *gmond* instance and the Ganglia Front End which obtains the federated data from *gmetad* and charts the data.

Our Architecture: 25 VMs each running the node software and one external client executing queries.

Each VM image was based on CentOS 5.6 and hosted a simple LAMP server and ran Wordpress which was subjected to minor artificial load using the Apache JMeter load testing tool. Each monitoring tool collected the same data: CPU load, memory usage, disk usage and the number of processes used. The bandwidth used by monitoring functions was collected using *ntop*. Monitoring traffic is potentially costly and a key criteria of cloud monitoring is behaviour with reduces both performance and financial costs. While each monitoring solution collects a vast array of different data, our tests attempt to collect the most basic metrics which all three solutions collect and store in almost identical ways. Each of the three systems can be configured to collect substantial volumes of additional data, however the exact nature of the data varies as does the manner in which data is collected, represented and transmitted vary considerably preventing any direct comparison. While these metrics will result in insubstantial volumes of traffic the degree of overhead which is required to acquire these values will be indicative of the degree of overhead which will be encountered in the case of high volumes of data. By requiring additional dedicated monitoring servers which collect data at a central point (or set of centralised points) Ganglia and Nagios will inevitably require greater bandwidth to communicate data to those servers as opposed to our architecture which does not forward data to beyond its composite members. This proof of concept demonstrates that for large scale cloud systems centralised monitoring systems are inferior and will result in significant communication between and out with clouds with will impact significant performance penalties and potentially costs.

6.3 Results

6.3.1 Nagios

The average traffic produced by Nagios per VM per hour was 51.23 Kilobytes. In addition to the traffic sent from the VM to the monitoring server, the monitoring server sent on average 69.1 Kilobytes to each VM per hour. The traffic sent to the VM primarily consisted of ICMP and SSH traffic which represents Nagios performing checks to determine availability and to perform a remote login to obtain data from the VM. Traffic sent from outside a cloud to a cloud virtual machine is not charged for by any IaaS provider and as such is not a significant limiting factor. The checks which were performed by Nagios represent the most basic and the default set of service checks. In order to obtain the four metrics, which are under a Kilobyte for an hours worth of data, Nagios imparts a significant overhead. This overhead is not specific to these four metrics and represents the same overhead which is present with all Nagios monitoring functions. In a real world context, Nagios will perform a far wider range of checks and obtain a far greater number of metrics and remain encumbered by the same degree overhead. While the traffic shown here is trivial in size, the volume of overhead is significant and is over 95% the size of the collected data. This level of overhead in unacceptable for a cloud computing monitoring system and would inevitably result in increased costs.

6.3.2 Ganglia

Similar to Nagios, Ganglia produces extensive overhead when collecting the four metrics. Ganglia produces 169.2 Kilobytes of traffic in its attempts to obtain the four metrics. This is again unacceptably high overhead to obtain small volumes of data when bandwidth is a metered service. Unlike Nagios, Ganglia pushes all of the collected data from client to server and as such all traffic Ganglia produces is traffic which leaves the cloud. This behaviour results in excessive outbound traffic which when is proportionally far greater than the data which is of value.

6.3.3 Prototype Implementation

Unlike for Ganglia and Nagios, the value which each of the graphs show is not for the data from each host. As only a root node responds to queries the values shown represent the bandwidth required to transmit the entire set of values for every node. Thus the prototype implementation on average requires 26 Kilobytes of bandwidth to return the data for every node. Therefore the prototype requires less bandwidth to transmit every metric than Ganglia and Nagios require to transmit a single VM's metrics. This is in part aided by data being transmitted to the client with very little overhead but also in part due to the maintaining of state. Each request issued by the client returns only the metrics which have changed since the last query. This behaviour can be seen in figure 6 where there are peaks in bandwidth use which represents significant changes in state meriting additional bandwidth use. Nagios and Ganglia however maintain no state and retransmit value even if there has been no change. This minimum of overhead allows for an entire representation of a set of value for an entire system with very little wasted bandwidth making it highly efficient for collecting data from a IaaS cloud.

6.4 Discussion

The results clearly show that our prototype is by far the most efficient of the three monitoring systems with regards to bandwidth usage. Both Nagios and Ganglia require a proportionally large amount of bandwidth to collect four small values. When this scales to significant volumes of data collection at faster frequencies the volume of data cause due to

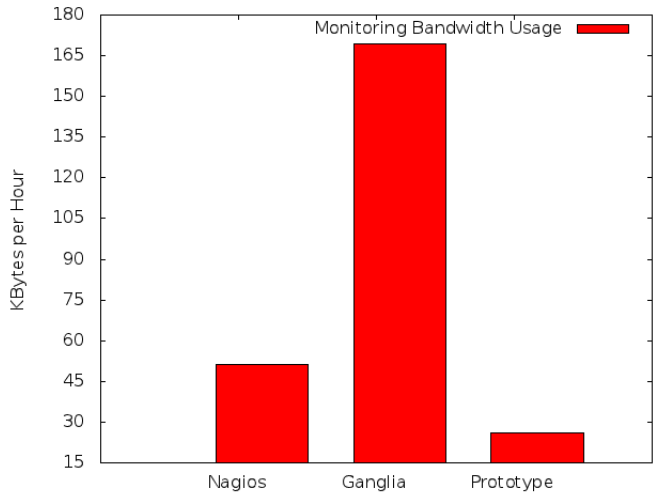

Figure 6: Average Monitoring System Bandwidth Usage Per Hour Per Host

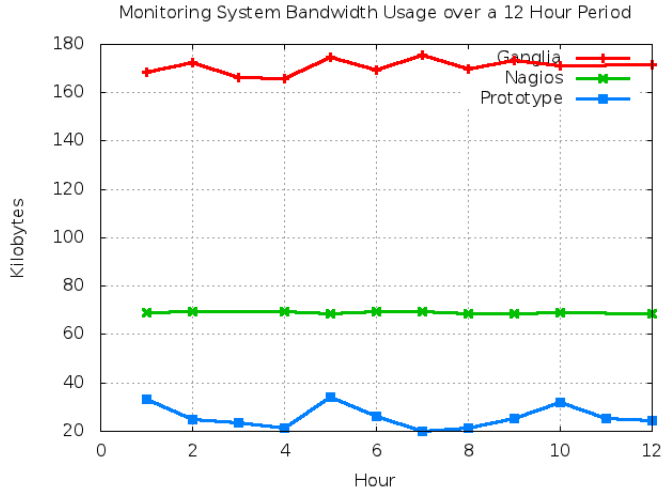

Figure 7: Monitoring System Bandwidth Usage Over 12 Hour Period

this overhead will be unacceptable for large scale cloud monitoring purposes. The prototype is sufficiently optimised for data transfer on the cloud such that it is able to represent the entire system's metrics in less data than the other two monitoring systems require for acquiring a single VMs metrics. This suggests purely from a data transfer prospective that our prototype implementation is a superior option for cloud based data collection. Notably during the setup of this evaluation our prototype required a minimum of configuration. Other than creating the VM image, setting the location of the bootstrap service and instantiating the VMs, no human interaction was required. Nagios required a separate definition on the server for each node and a configuration file specifying which metrics to collect and each client required configuration to allow the metrics to be obtained by the server. Ganglia, while requiring less configuration, still required configuration to locate data sources and to provide each data source with an identity. This was an incredibly time consuming activity, and while most of this work could be scripted it still presents unnecessary complexity and effort which is unacceptable for a large scale cloud system.

7. FUTURE WORK

7.1 Longevity Prediction

The ordering of heaps within the architecture is based upon a prediction of VM longevity. This prediction is currently based upon system load.. The working assumption of the prediction is that a highly loaded system (a further assumption is made that a loaded system is performing meaningful work) will continue to operate. Thus if a system is idle it is considered to be more likely to be terminated. This naive assumption does not necessarily hold true for all cases and can in some cases be detrimental to system performance as higher loaded hosts are put under further load. This simple prediction algorithm is to be replaced with an algorithm which considers multiple factors and performs a more complex analysis in order to improve accuracy.

7.2 Location Awareness

In our current implementation of the architecture, nodes are assigned to the smallest heap upon connection. This is a naive behaviour which can create suboptimal heaps where nodes are logically or physically distant. Location is especially significant in cloud computing as costs can be incurred for the transmission of data out with a data center of cloud provider. To mitigate the costs and latencies associated with distance nodes should be assigned a location within the architecture based upon factors relating to it's location. Conceptually this would allow for a heap containing nodes which are hosted by a single cloud provider, hosted within the same data center or are logically in close proximity with regards to an application or service.

7.3 Reduction of Load on Root Nodes

Root nodes are responsible for significant data collection and storage functions. In a system where every node is highly loaded it is unlikely that any node will have the required capacity to act as an efficient root node. This can be mitigated by increasing the number of heaps within the system to reduce the size of each heap, however this does not entirely eliminate the problem. To significantly improve

root node performance a form of load balancing strategy, where by the root of a heap is not a single node is required. While maintaining the conceptual architecture, having multiple nodes share the responsibilities of being a root node for a single heap minimizes the data collection duties of each node.

8. CONCLUSIONS

The proposed architecture exhibits promising features which make it applicable for a wide range of purposes in large scale dynamic systems. While current monitoring infrastructures require significant additional resources, configuration and management our architecture operates with a minimal of human involvement over the existing infrastructure. The semantic based holistic view it provides has a number of potential future applications which will be further investigation upon. As the scale, complexity and level of dynamic behaviors in cloud systems increase the ideas expressed in our architecture will become increasingly relevant.

9. REFERENCES

[1] Anderson, D.P. Local Scheduling for Volunteer Computing. pages 1 –8, march 2007.

[2] Birman, K.P. and van Renesse, R. and Vogels, W. Navigating in the storm: using Astrolabe for distributed self-configuration, monitoring and adaptation. pages 4 – 13, june 2003.

[3] Dan Brickley, R.V Guha, Brian McBride. RDF Vocabulary Description Language 1.0: RDF Schema.

[4] Hoffa, C. and Mehta, G. and Freeman, T. and Deelman, E. and Keahey, K. and Berriman, B. and Good, J. On the Use of Cloud Computing for Scientific Workflows. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, volume , pages 640 –645, dec. 2008.

[5] Kephart, J.O. and Chess, D.M. The vision of autonomic computing. *Computer*, 36(1): 41 – 50, jan 2003.

[6] Kozuch, Michael A. and Ryan, Michael P. and Gass, Richard and Schlosser, Steven W. and O'Hallaron, David and Cipar, James and Krevat, Elie and López, Julio and Stroucken, Michael and Ganger, Gregory R. Tashi: location-aware cluster management. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 43–48, New York, NY, USA, 2009. ACM.

[7] Matthew L Massie and Brent N Chun and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004. .

[8] Nagios Enterprises. Maximizing Nagios XI Performance, `http://exchange.nagios.org/directory/Documentation/Nagios-XI-Documentation/Maximizing-XI-Performance/details`.

[9] Nagios Enterprises. Nagios, `http://www.nagios.org/`.

[10] Northrop, L. and Feiler, P. and Gabriel, R. P. and Goodenough, J. and Linger, R. and Longstaff, T. and Kazman, R. and Klein, M. and Schmidt, D. and Sullivan, K. and Wallnau, K. Ultra-Large-Scale Systems - The Software Challenge of the Future. Technical report, Software Engineering Institute, Carnegie Mellon, June 2006.

[11] Parashar, Manish and Hariri, Salim. Autonomic Computing: An Overview. In BanÃćtre, Jean-Pierre and Fradet, Pascal and Giavitto, Jean-Louis and Michel, Olivier, editor, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 97–97. Springer Berlin / Heidelberg, 2005.

[12] Rohloff, Kurt and Dean, Mike and Emmons, Ian and Ryder, Dorene and Sumner, John. An Evaluation of Triple-Store Technologies for Large Data Stores.

[13] The Eucalyptus Community. The Eucalyptus Open Source Cloud Platform.

[14] The Ganglia Project. `http://ganglia.info/`.

[15] University of St Andrews. St Andrews Cloud Computing Co-laboratory: Collaborative Research in Cloud Computing.

[16] W3 Consortium. SPARQL Query Language for RDF.

[17] W3 Consortium. The Resource Description Framework.

[18] W3 Consortium. Turtle - Terse RDF Triple Language.