Self Managing Monitoring for Highly Elastic Large Scale Cloud Deployments

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

Adam Barker
Univeristy of St Andrews
adam.barker@st-andrews.ac.uk

ABSTRACT

Infrastructure as a Service computing exhibits a number of properties, which are not found in conventional server deployments. Elasticity is among the most significant of these properties which has wide reaching implications for applications deployed in cloud hosted VMs. Among the applications affected by elasticity is monitoring.

In this paper we investigate the challenges of monitoring large cloud deployments and how these challenges differ from previous monitoring problems. In order to meet these unique challenges we propose Varanus[1], a highly scalable monitoring tool resistant to the effects of rapid elasticity. This tool breaks with many of the conventions of previous monitoring systems and leverages a multi-tier P2P architecture in order to achieve *in situ monitoring* without the need for dedicated monitoring infrastructure.

We then evaluate Varanus against current monitoring architectures. We find that conventional monitoring tools perform acceptably for small, non changing cloud deployments. However in the case of large or highly elastic deployments current tools perform unacceptably incurring increased latencies, high load and slowed operation necessitating that a new, alternative tool be used. Further, we demonstrate that Varanus maintains low latency monitoring with limited demand upon resources, even during during periods of high elasticity.

1. INTRODUCTION

Cloud computing has made computing at scale available to all through a pay-per-use model. While previously the deployment of large scale systems required significant funds and resources, it is now feasible for individuals to deploy a large number of transient cloud virtual machines (VMs) for a short period of time. This new-found availability has put scalability at the forefront of system design.

[1]Varanus is the genus name of the monitor lizard

DIDC 2014, June 23, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2913-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2608020.2608022 .

A deployment of cloud VMs differs from a conventional physical server deployment [12] [14]. Cloud deployments are inherently elastic, which is a key benefit of cloud computing [10]. Elasticity entails the ability for resources to be provisioned and released as is necessary. The result is that a cloud deployment can change entirely in composition, scale and function in a very short period of time. This propensity for change invalidates many previous system architectures and requires that new software systems be designed to tolerate the properties of elasticity.

Among the systems affected by elasticity is systems monitoring. In all but the most trivial deployments, monitoring is essential. Allowing for the detection and investigation of failure, misconfiguration, performance and other issues, monitoring is a key part of the design, implementation and maintenance of software systems. In the case of large scale systems, monitoring is crucial in order to understand complex and emergent system properties. Monitoring at scale, is however, a significant challenge [16] [22]. It requires the collection, storage and processing of a large volume of information from a vast range of sources. Not only is this process data intensive but it is also computationally expensive. When these challenges are compounded by rapid elasticity the task puts considerable strain on current tools.

Current monitoring tools originate from previous paradigms of computing including Cluster [19], HPC [13], Grid [20] and conventional server computing [15], domains which have differing requirements to cloud computing [4] [17]. Despite the differences, many of these tools have been retrofitted to monitor cloud deployments and do not make provisions for many of cloud computing's unique properties [1]. As a result of this, many existing tools only function well for small deployments which exhibit limited elasticity [11]. When monitoring larger, elastic deployments many existing tools exhibit significant latencies and overheads which render them ineffectual copperegg. In addition to the operation of monitoring being costly, these architectures also result in an overhead of additional infrastructure. Furthermore, this has the potential to incite a classical problem: who watches the watchers. Current common practice is to augment current tools with a complex array of plugins and configuration or to build bespoke tools. This requires significant development time and expense and is unavailable to many cloud users.

We therefore contend that it is necessary to design a new monitoring tool for large scale cloud deployments, which abandons conventional monitoring architectures. This paper attempts to quantify the challenges of monitoring a cloud deployment, primarily the challenges arising from rapid elas-

System	Origin	Architecture
Nagios	Server Monitoring	Hierarchical Pull
Ganglia	HPC	Hybrid Push/Pull
CloudWatch	Amazon	Abstracted Push
Collectd	UNIX monitoring	Hierarchical Push
Big Brother	Server Monitoring	Hierarchical Pull

Figure 1: Comparison of common cloud monitoring tools

ticity and from scale. To this end we describe the effects of these properties and propose a series of strategies in order to mitigate these properties. We then propose Varanus, a new monitoring system utilising these strategies in order to provide robust and reliable monitoring for large scale cloud deployments.

The remainder of this paper is structured as follows: Section 2 describes the current systems which are frequently used to monitoring cloud systems. Section 3 describes the motivations for the development of a cloud based monitoring system. Section 4 describes our architecture and it's implementation. Sections 5 and 6 evaluate our architecture against current architectures.

2. RELATED WORK

There are a number of specific monitoring systems which are commonly used for monitoring cloud systems, the most prominent of which are summarised in Figure 1, in detail these are:

Nagios [15] is the de facto standard Open source monitoring system. It provides a wide range of host and network monitoring plugins allowing for the monitoring of a considerable range of infrastructure and software. Architecturally, Nagios uses a central server to poll monitored servers either directly of through an intermediary server. Nagios uses a range of custom and standard protocols to interact with monitored servers and relies upon an SQL database for storage.

Collectd [3] is a UNIX data collection daemon which provides an efficient mechanism for pushing monitoring data to a multicast group, server or server hierarchy. Collectd utilises its own binary protocol for compact data encoding and is frequently used alongside RRDTool to store the collected data. It serves as the basis for several cloud based monitoring solutions including Rightscale Monitoring [6].

Ganglia [13] is a scalable resource monitoring primarily intended for monitoring HPC, cluster and grid deployments. Ganglia utilises a push mechanism to federate monitoring state and then a hierarchical pull mechanism to aggregate federated state to a top level server. Ganglia utilises XML and XDR for data representation and relies upon RRDTool for data storage.

CloudWatch [18] is Amazon Web Service's monitoring as a service tool. The inner workings of CloudWatch are uncertain due to it's proprietary nature. CloudWatch allows state can be pushed via a rate limited API which can then be accessed via a web interface. Amazon abstracts over the underlying monitoring resources however given the tool's ability to scale there is inevitably a significant pool of resources underlying it.

2.1 Architectures

It is common practice for stakeholders to deploy a monitoring system based upon tools designed for an alternative domain. These tools are deployed either as part of a larger system or as a full system in an of themselves. Each of these systems implements one of the following architectures [24]:

- Flat pull model. This is the architecture employed by Nagios, The Windows Management Instrumentation, Icinga, Xymon and Cacti. A central server polls a set of monitored servers according to a schedule that it computes when clients leave and join. The schedule can be adjusted to poll metrics at a different rate if necessary.
- Hierarchical pull model. A modification of the above architecture, the central server polls a hierarchy of monitoring servers which in turn poll a pool of monitored servers.
- Hierarchical Push Model. This is similar to the architecture employed by Ganglia and Collectd. Agents push monitoring state at their own volition to one of a set of servers which in turn pushes the collected state to a central server.

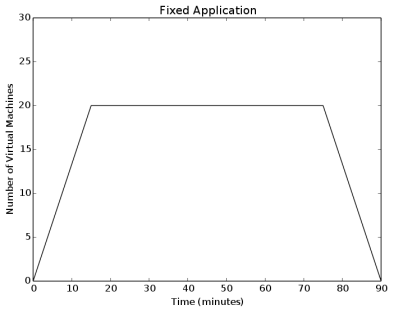
3. ELASTICITY AND MONITORING

Elasticity is the ability of a deployment to adapt to changing requirements by allocating or deallocating resources. Elasticity allows a deployment to adapt to meet new demands by changing the number of VMs and by changing the types of VMs. Both of these properties are challenging to monitoring tools. The latter presents a logical challenge: how to enumerate and understand change. The former, however, presents a fundamental challenge to distributed applications and can potentially inhibit monitoring.

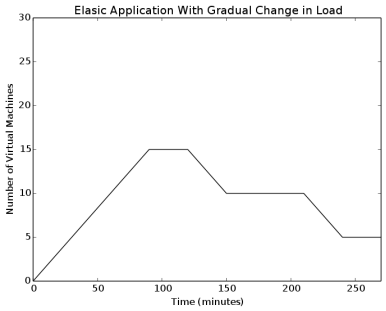
Any rate of change of deployment membership is potentially problematic as many current tools are far from autonomic and require manual configuration to add and remove monitored hosts [1]. When new VMs are instantiated they must be bootstrapped to join the deployment, which is often a costly operation. When VMs are terminated any shared state or workloads must be redistributed among the remaining VMs and there is the risk of data loss if redistribution cannot occur before the VM is terminated. If a deployment is highly elastic and is frequently undergoing change then the effects of instantiation and termination constantly occur.

The implications of this aspect of elasticity are usually not severe. The reason for this is that instantiation and termination occur based upon the needs of a single application within the deployment. In current common use cases, instantiation and termination occurs based upon the load encountered by a web application. As load increases additional VMs are provisioned and as load subsides those additional VMs are terminated. As there is a direct correlation between an application and VM provisioning there is no sudden termination. Termination occurs when it is convenient for the application; when load has subsided and less work is being performed. This means that for the web application the effects of elasticity are primarily positive. For applications running in the VMs other than the web application the effects are potentially negative.

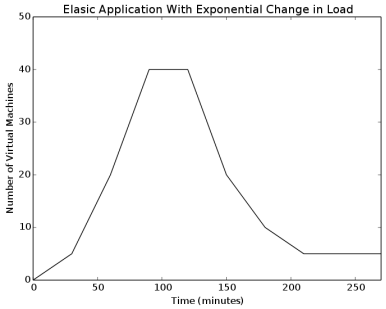
For applications running alongside the main application (in this use case, a web application) the instantiation and



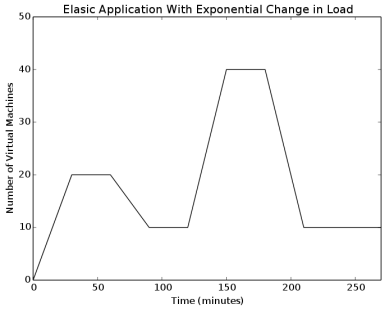
(a) Fixed Architecture



(b) Elastic Architecture With Gradual Change in Load



(c) Elastic Architecture With Exponential Change in Load



(d) Elastic Architecture With Random Change in Load

Figure 2: Autoscaling Cloud Deployments With Varying Levels of Load

termination of VMs does not occur when most convenient. The requirements of 'secondary' applications, such as monitoring software, are ignored. The implications is that software running alongside the primary application must be able to handle the sudden addition and removal of VMs in as graceful a manner as possible.

3.1 Quantifying Elasticity

In order to design applications which tolerate elasticity, it is necessary to understand the patterns of VM instantiation and termination that commonly occur. Elasticity in most common use cases is based upon the use of a load balancer. The load balancer handles incoming requests and spawns additional VMs if the volume of requests exceeds a given threshold and then terminates VMs when demand subsides.

In order to examine elasticity, we deployed a simple web application on Amazon Web Services which made use of the load balancing autoscale feature. As request rates increase beyond a standard set of thresholds the load balancer instantiates additional VMs to handle the load. Load was generated using the Apache JMeter Web Server testing tool [5] over a 4 hour period, according to three access patterns: a gradual step up and step down in load, an exponential step up and logarithmic step down in load and a randomly generated change in load. Additionally an application which does not autoscale and remains at a fixed size is deployed as a comparison. Figure 2 shows the number of VMs that the load balancer instantiated to meet the demands of load.

These patterns of elasticity represent some of the typical patterns that cloud deployments will encounter. The relatively sudden increases and decreases present significant challenges for the software running along side the primary application. These applications, which include monitoring tools will suffer from the negative effects of elasticity. The patterns of elasticity described in this Section have been adapted into models which are used to evaluate monitoring architectures in Section VII to determine how well they handle this aspect of rapid elasticity.

4. VARANUS

Varanus is a scalable distributed monitoring tool which is resistant to rapid scalability. In lieu of a conventional hierarchy, Varanus employs a layered gossip architecture [9] with a novel grouping scheme which provides efficient data collecting and analysis over existing resources. We propose Varanus as a means to handle the challenges presented in the previous Sections, and as an alternative to previous monitoring paradigms which are not well suited for monitoring large scale cloud deployments. This Section describes the design of Varanus.

4.1 Communication

In lieu of a conventional unicast hierarchy, communication of monitoring data is achieved via the use of a layered probabilistic multicast or gossip protocol. In large scale cloud deployments individual VMs operate under a range of computation and communication constraints. By distributing the computational complexity of an operation over the system, gossip protocols offer a means to develop mechanisms better suited to large scale systems. Gossip protocols have

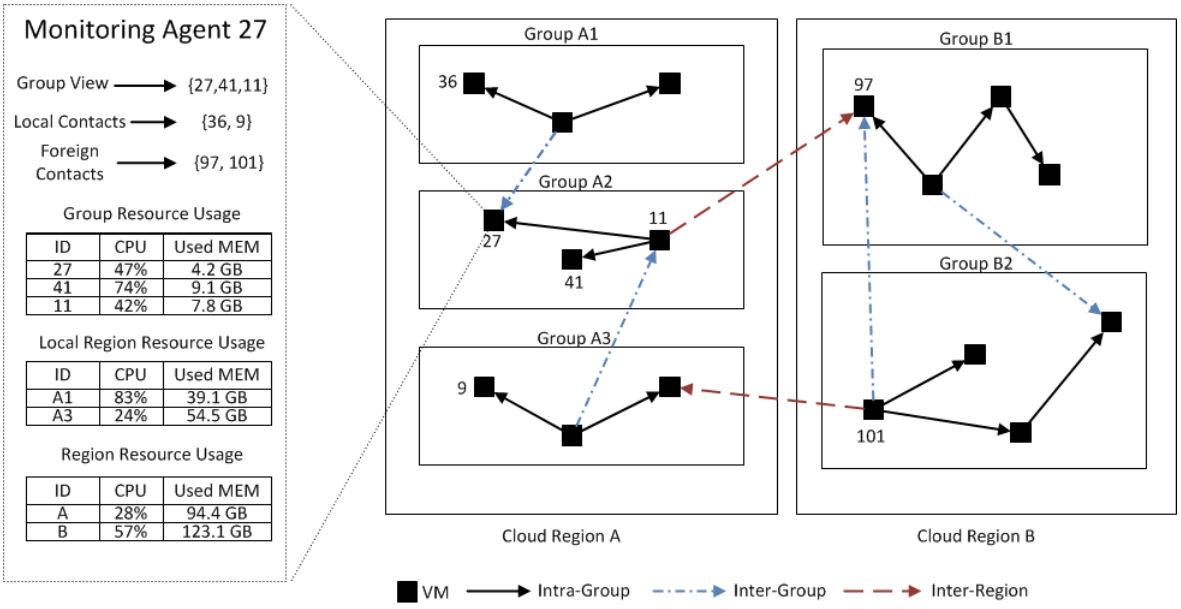


Figure 3: Architectural Overview of Varanus. The panel to the left represents the internal state stored at a single VM. This is the state propagated via the gossip protocol. The panel to the right represents VM groupings as described in section 4.3. The varying types of line denote communication between groups as defined by Section 4.2.

been demonstrated to be effective mechanisms for providing robust and scalable services for distributed systems including information dissemination [2], aggregation [7] and failure detection [21].

The basic operation of the Varanus gossip protocol consists of the periodic, pairwise propagation of state between two processes. This mechanism underpins the data collection and agreement protocols which support monitoring functions. Each monitoring agent participates in a gossip based overlay network. Using this overlay monitoring agents propagate and receive state from other, nearby, agents. This is achieved by performing a pull-push operation with neighbouring correspondents. The rate of dissemination of data from a single process to all other processes can be described by the following equation:

$$S_{t+1} = T_{interval} \times Fanout \times \frac{S_t X_t}{n} \quad (1)$$

where S is the number of susceptible processes (those which have not yet received the information), X is the number of infected processes (those which have received the information), n is the number of processes and t is the current time step. Therefore, the delay in propagating information can be greatly reduced by decreasing the interval at which communication occurs (thus increasing the frequency) and by increasing the fanout value (thus increasing the number of targeted VMs).

In addition to this mechanism, preferential VM selection is used to reduce the delay in propagating state. VMs are selected based on a weighting scheme which uses round-trip time estimates in order to select VMs which are topologically closer. Each round of gossip is spatially weighted according to the scheme proposed in [8], using RTT as a distance metric in order to propagate updates to all nodes within distance d within $O(\log^2 d)$ time steps.

This scheme results in increased memory usage and constant background communication but achieves rapid state propagation and resilience to elasticity and failure. Within a cloud where there is high bandwidth, low latency and no service metering this trade-off is acceptable.

4.2 Communication Hierarchy

In order to best exploit the topology of IaaS clouds Varanus exhibits different behaviours at each level of the gossip hierarchy. The rationale for this hierarchy is rooted in the differences between intra and inter cloud communication. Within IaaS environments there are high bandwidth, low latency and unmetered network connections. This is true of virtually all cloud providers. It is also true of any private cloud with a public network between cloud regions. This environment lends itself to the use of an unreliable protocol for rapid and near constant state propagation. Between cloud regions this is not as feasible, costs arising from latency and bandwidth metering force communication to be performed in a slower, more reliable fashion.

The gossip protocol described in Section 4.1, is applied at every level of the hierarchy. What differs between each level is the information which is communicated and the frequency at which communication occurs. There are three levels of the hierarchy as shown in Figure 3

1. Intra Group: communication between monitoring agents within the same group. This occurs at a near constant rate. Each time a state change, deemed notable by the monitoring agent, occurs the correspondent propagates the new state to it's group. At this level of granularity, the full state stored by the monitoring agent is propagated to its neighbours.

2. Inter-Group: communication between monitoring agents in different groups within the same region. This occurs at a frequent but non constant rate. Periodically state is propagated to external groups according to a shifting interval. At this level, only aggregated values for the region resource usage and a small subset of local contacts and foreign contacts are propagated.

3. Inter-Region: communication between monitoring agents in different different cloud regions or data centers. This occurs proportionally to the inter-group rate. At this level an aggregate value for the entire region and subsets of the local and foreign contacts are propagated between regions.

4.3 Virtual Machine Grouping

One of the most common use cases of a monitoring system is as follows: A user requests information regarding a server for example: CPU usage or Apache response time. The monitoring tool then fetches the requested information from it's datastore. If sufficiently recent information is unavailable it will obtain it from the pertinent VM. The monitoring tool then visualises the information in an appropriate way and the user makes a judgement based upon that information and then, if necessary, acts appropriately to modify the system.

In this use case there are potentially multiple interactions between the monitored server, the datastore, the front end of the monitoring tool and the user. Dependant upon the specifics of the monitoring tool, this can result in significant overhead in order to provide the user with even the most trivial information. In an autonomic context, this use case changes as there is no longer a user making the decision, in their place is a software agent. This raises the question as to the placement of the equivalent computation. Using the mechanism describe above, any idle or under utilised VM could be identified and tasked with the operation. Doing so would, however, incur many of the same inefficiencies as per the human orientated computation. Rather, it is preferable to reduce the overall volume of data movement in order to perform decisions faster, with as fresh information as is feasible.

In order to achieve this, we propose a novel grouping mechanism. While a layered gossip approach reduces communication overhead when compared against a flat approach, latency and the rate at which information is requested can be reduced through the use of appropriate grouping. In Varanus, this grouping is achieved though the use of feature vectors. Upon instantiation a VM computes a feature vector which describes the following, in order of importance:

1. Location. The location of the virtual machine down to the smallest unit. The exact nomenclature is cloud dependant but in general terms, this will correspond to a data center, availability zone, region or other abstraction.

2. Primary software deployed in the VM. Software that the VM was deployed in order to provide, including but not limited to web servers, databases, in memory caches, distributed computation tools and so forth.

3. Seed information. Information provided to the VM at boot time including but not limited to the id of the stakeholder who instantiated the VM, hostnames and addresses of common resources and user provided annotations.

4. Secondary software, other than monitoring tools. Software which supports the primary application or otherwise adds additional functionality.

This information is represented using a weighted 4 dimensional feature vector which describes the above attributes. The attributes are weighted according to the impact they have on communication. Location serves as the most pertinent factor as it is largely responsible for determining latencies and other costs. The other factors imply relations between VMs in the forms of shared purpose or use of shared resources. This suggests the likelihood that information collected from a VM will be relevant to a similar VM.

Upon instantiation a VM computes its own feature vector and obtains a list of groups from a bootstrap node. The VM then compares its own feature vector against an averaged feature vector describing the properties common to the group. The VM joins the group which is deemed the most similar according to an algorithm based upon the k-Nearest Neighbours algorithm [23]. If according to the KNN algorithm, a VM falls within a significant distance of multiple groups the VM can join all of the related groups.

This grouping mechanism has the result of placing related VMs within logical proximity. According to the above communication scheme, data travels the least distance to related VMs allowing analytics and autonomic decision making to occur with reduced latency and as close to the pertinent data as possible.

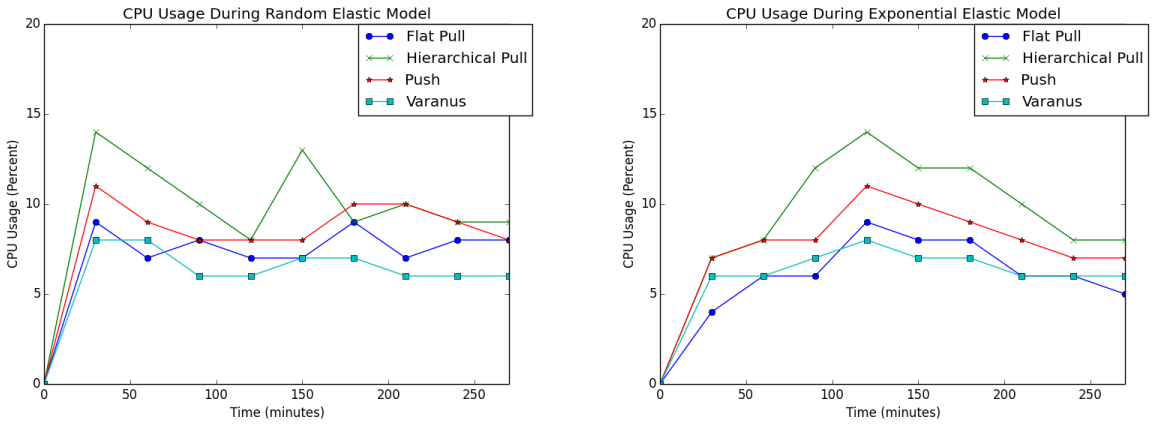
4.4 Interaction

Unlike conventional client-server based monitoring tools Varanus has no single monitoring server and therefore no single point of communication. Queries to obtain monitoring data are routed and results fetched according to the gossip scheme described above. In order for an external user to communicate with a Varanus deployment a node must serve as a gateway to accept, route and return the response to requests. Any node can act in this capacity. A variation of the group allocation algorithm is used to select a optimum node such that the gateway node is as close to the relevant data as possible. This ensures that requests for monitoring data can be fulfilled in the shortest possible time.

5. EXPERIMENTAL EVALUATION

Our experiments focus upon elasticity and scalability and investigate how the proposed architecture compares against previously established architectures. The range of monitoring tools that are available do not lend themselves to easy investigation. The vast disparities in APIs, data representation formats, languages and protocols found in current monitoring tools prevents unbiased comparisons of the underlying architectures. Therefore, in order to evaluate the common monitoring architectures we developed a series of purpose build tools which implement the relevant architectures. This allows for a like for like comparison regarding the properties of the monitoring architecture.

Our experiments were conducted on Amazon EC2, using a test bed of 200 VMs. Each VM was a m1.medium instance with 64 bit CPU, 3.75GiB memory and around 100Mbps network bandwidth. Varanus, and the other monitoring architectures were implemented in Java using ZeroMQ to pro-



(a) Random Elasticity

(b) Exponential Elasticity

Figure 4: CPU Usage During Elasticity

vide message passing and Google Protocol Buffers to provide data encoding.

Our evaluation examines four monitoring architectures. A flat pull architecture, a hierarchical pull architecture, a hierarchal push architecture and the architecture of Varanus. These architectures are described in section 2 and 4 respectively.

6. RESULTS

6.1 Resource Usage During Elasticity

The graceful handling of VMs joining and leaving the system is essential to ensure continuous monitoring undistributed by change. Figures 4(a) and 4(b) show CPU usage of each monitoring system as a percentage of overall system CPU whilst, respectively, an exponential and random elasticity occurs.

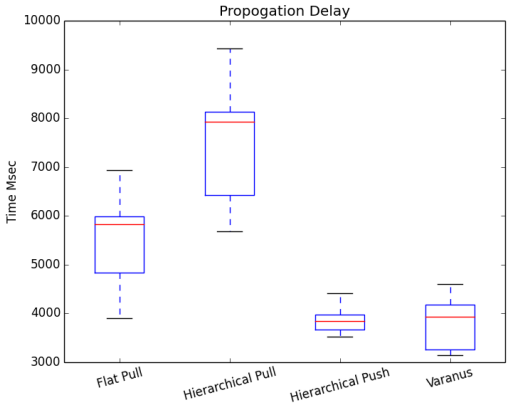
It is clear that in both cases that elasticity produces high resource consumption in the pull models in both the exponential and random case. During a period where VMs are being instantiated monitoring servers must handle joins and recompute the polling schedule while still performing regular polls. At its peak in the exponential case, the flat pull monitoring scheme accounts for 9% of the entire deployment's CPU usage. Meanwhile, due to additional resource being dedicated to monitoring, the hierarchical pull accounts for 14% of system wide CPU usage. For a function other than the primary function of the deployment, that level of resource usage in unacceptable and has the potential to interfere with the deployments primary application. The push model improves upon this performance by consuming 11% of system wide CPU at peak time. This resource usage is primarily due to monitoring servers having to handle a sudden increase in the number of join messages. Varanus meanwhile consumes 6% of CPU resources during the majority of the exponential case, encountering a peak of 8% usage at the greatest point of elasticity. Both the random and exponential elasticity cases shows that the pull and push systems have more conservative resource demands while the system is smaller and whilst it experiences less elasticity. At the start and end of the exponential test and intermittently during the random test resource usage for the flat pull reaches as low as 5%, the hierarchical pull, 6% and the hierarchical push 4%. Varanus maintains around a constant 6% resource usage throughout.

6.2 Propagation Delay

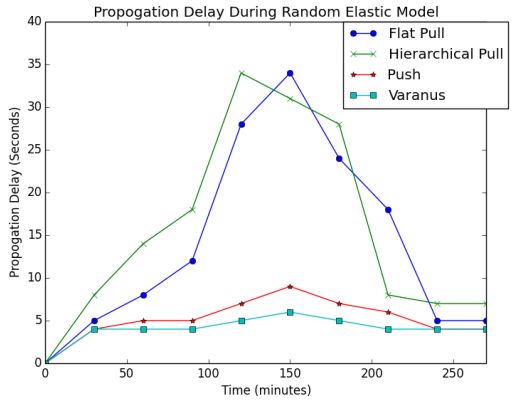
Figure 5(a) shows the time required for state from a monitored VM to be made available to an external consumer. The comparatively wide range of delays encountered by the two flat models is due to the manner in which polling occurs. With a polling interval of 5 seconds there is an explicit latency until fresh data is obtained. The delay in obtaining fresh data is therefore dependant upon when during that polling interval the data is collected. The hierarchical push model and Varanus, fair better with around a 50% reduced propagation delay. This reduction is due to the push mechanism ensuring that fresh data is transmitted when it becomes available, eliminating a polling latency. The discrepancy between the range of delays encountered by the push model and Varanus is due to the nature of the gossip protocol. The delay in Varanus is dependant upon the gateway node's distance from the producer. The closer the gateway is to the producer, the faster the rate of propagation. In the mean case the gateway is sufficiently close to the producer such that the data is received in a small number of gossip rounds making the mean delay comparable to the mean delay encountered by the hierarchical push. It the worst case, Varanus yields a 5% slowdown against the hierarchical push, while in the best case yields a 12% improvement against the hierarchical push. This is not, however, the primary use case of Varanus. Instead, Varanus concerns itself with reducing inter-deployment propagation delay in order to support autonomic monitoring and places less emphasis on delivering state to human users or external consumers as per previous tools.

6.3 Propagation Delay During Elasticity

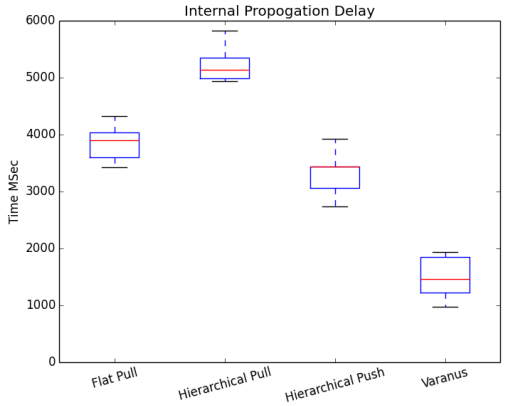
The previous test was repeated whilst undergoing each of the elasticity models. Figure 5(b) shows the propagation delay over time while each architecture underwent the random elasticity model.



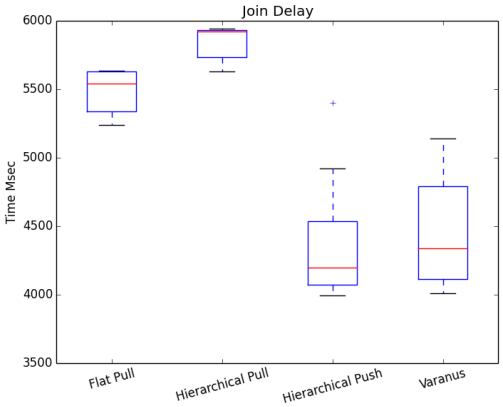
(a) Propagation Delay



(b) Propagation Delay During Random Elasticity



(c) Internal Propagation Delay



(d) Join Delay

Figure 5: Monitoring Latency Results

In the pull systems, propagation delay severely increases to up to around 30 seconds for both at their peak. The repeated computation required to successfully propagate monitoring state fails to be completed on time and the high load encountered by monitoring components reduces the responsiveness of the monitoring architecture further. In the push system there is a moderate significant increase delay during high elasticity, around a 100% increase from when the system is stable in the worst case taking around 8 seconds to propagate new state. Varanus meanwhile maintains a near constant propagation delay of around 4 seconds with a peak of 6 seconds when the system is at high elasticity.

6.4 Inter-Deployment Propagation Delay

Inter-Deployment Propagation Delay, that is the time taken to propagate monitoring state to VMs within the cloud deployment. This is desirable for a number of use cases as monitoring state is equally valuable to the software operating in VMs as is to external users. Access to monitoring state allows applications to alter their behaviour, predict load, eliminate redundancy and a number of other beneficial autonomic applications. Without access to monitoring data, it is difficult to implement any autonomic behaviours. As shown in figure 5(c), Varanus achieves significantly reduced delay compared to current architectures. Monitoring state is propagated to related and nearby hosts and the delay incurred by the push and pull systems is only experienced in Varanus if an entirely unrelated VM (according to the group mechanism) requests monitoring state. Therefore, in its best case, Varanus has up to a 300% faster internal propagation rate compared against other architectures. This makes Varanus far superior for providing state to autonomic applications or other software where fresh data is beneficial.

6.5 Join Delay

In a highly elastic system the join operation will be performed frequently. The longer the join operation the longer there is period where no monitoring of the joining VM occurs. Figure 5(d) shows the time required to complete the join operation for each of the monitoring architectures. In the case of the push model there is no formal join operation, joining is achieved by simply communicating with the hierarchy. This is similar to join operation of Varanus, but this operation is preceded by the grouping mechanism, accounting for the 8% increase in time between the mean join time of Varanus and the push model. Both the push model and Varanus achieve significantly faster joins than the pull systems. The pull systems require the monitoring servers to be notified, the polling schedule to be updated and a polling cycle to occur prior to state being made available.

7. CONCLUSION

Cloud monitoring is a significant challenge. Monitoring at scale and monitoring a constantly changing deployment is extremely costly. Despite the cost, it is necessary and it is therefore essential to develop tools better suited to cloud monitoring. Varanus, our proposed monitoring tool offers demonstrably better scalability and tolerance to rapid elasticity than other existing monitoring architectures. It is clear that if a cloud deployment is operating at scale with frequent changes occurring then existing architectures are not sufficient. Increased delay, high load and poor flexibility

inhibit more traditional centralised architectures when operating at scale and experiencing elasticity. It is clear that existing tools are well suited and sufficient for smaller deployments or mid size deployments which are not susceptible to rapid change. In the event of these two phenomena, more domain specific monitoring is required.

8. REFERENCES

[1] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio PescapÃÍ. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093 – 2115, 2013.

[2] Anwitaman Datta and Rajesh Sharma. Godisco: selective gossip based dissemination of information in social community based overlays. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 227–238, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Florian Forster. Collectd The system statistics collection daemon.

[4] I. Foster, Yong Zhao, I. Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, 2008.

[5] Emily H Halili. *Apache JMeter: A Practical Beginner's Guide to Automated Testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[6] Rightscale Inc. Rightscale monitoring system. `http://support.rightscale.com/12-Guides/RightScale_101/08-Management_Tools/Monitoring_System`, 2014.

[7] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, August 2005.

[8] David Kempe, Jon Kleinberg, and Alan Demers. Spatial gossip and resource location protocols. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC '01, pages 163–172, New York, NY, USA, 2001. ACM.

[9] A-M Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14(3):248–258, 2003.

[10] Ali Khajeh-Hosseini, David Greenwood, James W Smith, and Ian Sommerville. The cloud adoption toolkit: supporting cloud adoption decisions in the enterprise. *Software: Practice and Experience*, 42(4):447–465, 2012.

[11] Jonah Kowall. Got nagios? get rid of it. `http://blogs.gartner.com/jonah-kowall/2013/02/22/got-nagios-get-rid-of-it/`.

[12] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. Nist cloud computing reference architecture. *NIST special publication*, 500:292, 2011.

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

[14] Michael Armbrust and Armando Fox and Griffith, Rean and Anthony D. Joseph and Randy Katz and Andy Konwinski and Lee, Gunho and Patterson, David A. and Ariel Rabkin and Ion Stoica and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. (UCB/EECS-2009-28), Feb 2009.

[15] Nagios Enterprises. Maximizing Nagios XI Performance.

[16] 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.

[17] N. Sadashiv and S.M.D. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *Computer Science Education (ICCSE), 2011 6th International Conference on*, pages 477–482, 2011.

[18] Amazon Web Services. Amazon, inc.[online]. `http://aws.amazon.com/cloudwatch/`.

[19] Matthew J Sottile and Ronald G Minnich. Supermon: A high-speed cluster monitoring system. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 39–46. IEEE, 2002.

[20] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A grid monitoring architecture, 2002.

[21] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, London, UK, UK, 1998. Springer-Verlag.

[22] Jonathan Stuart Ward and Adam Barker. Semantic based data collection for large scale cloud systems. In *Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date*, DIDC '12, pages 13–22, New York, NY, USA, 2012. ACM.

[23] Xiaohui Yu, Ken Q Pu, and Nick Koudas. Monitoring k-nearest neighbor queries over moving objects. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 631–642. IEEE, 2005.

[24] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Computer Systems*, 21(1):163 – 188, 2005.