Ward and Barker *Journal of Cloud Computing: Advances, Systems and Applications*
DOI 10.1186/s13677-015-0041-9

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Cloud cover: monitoring large-scale clouds with Varanus

Jonathan Stuart Ward† and Adam Barker*†

Abstract

Cloud computing has quickly become the de facto means to deploy large scale systems in a robust and cost effective manner. Central to the maintenance of large scale systems is monitoring which allows for the detection of faults, errors and anomalies and the enacting of optimisation and corrective measures. Monitoring large scale systems is significant challenge requiring the low latency movement of large volumes of data and near real time analysis. This challenge is magnified by elasticity and other cloud properties which previous monitoring systems do not yet fully account for. In this paper we propose Varanus[1] a cloud aware monitoring tool that provides robust, fault tolerant monitoring at scale. We describe in detail the mechanisms which enable Varanus to function effectively and explore the performance of Varanus through a detailed evaluation.

Keywords: Cloud computing; Monitoring

Introduction

Monitoring is a fundamental part of designing and maintaining reliable and effective software systems. The data obtained from monitoring is invaluable allowing for the detection of error, misconfiguration and other interesting phenomena. As the Internet of Things, Big Data and pervasive computing become increasingly relevant, data has never been more valuable. However, as the size and complexity of systems has increased so to has the difficulty in collecting monitoring state. Cloud computing is a technology which underpins much of the so called data deluge and is an area where monitoring is a distinct challenge.

Prior to the advent of cloud computing, large scale systems were accessible only to organisations with the greatest means. Since 2007, cloud computing has brought large scale systems to the masses. Organisations and even individuals can now temporarily acquire significant compute and storage capacity via an elastic model for a fraction of the cost of a single physical server. Elasticity is the ability of a deployment to change in scale and composition in accordance with demand and is what sets cloud computing apart from grid, cluster and other earlier paradigms of distributed computing. The combination of elasticity and scale poses a series of challenges to a number of area, including monitoring.

Cloud management is a research topic which has received considerable attention. Common topics include configuration and change management, cost forecasting and architectures for deploying services and workloads to cloud services. Monitoring, an integral part of management has however received notably less attention. Monitoring cloud deployments is a challenge for a number of reasons. Scale necessitates that a monitoring tool must collect and analyse vast quantities of data in a timely manner, with minimum human intervention, while elasticity requires tolerance to all manner of change. Other areas of monitoring including failure detection, QoS and root cause analysis are also affected by elasticity.

Current monitoring tools fall into two broad categories: monitoring as a service tools which outsource data collection and analysis to a third party and legacy grid, cluster and enterprise monitoring tools such as Nagios, Ganglia and Cacti which have greater functionality but are ill suited to the requirements of cloud computing [1]. The former category of tool charge, typically, at a per host basis and can incur significant fiscal costs. Furthermore these tools transmit monitoring data across the Internet to the cloud provider, this potentially introduces security concerns and increases monitoring latency. The latter

*Correspondence: adam.barker@st-andrews.ac.uk
†Equal Contributors
School of Computer Science, University of St Andrews, St Andrews, UK

Springer

© 2015 Ward and Barker. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

category of tools are ill suited to cloud computing due as they lack awareness of cloud properties, as such they conflate termination with failure, are ill suited to handling elasticity and have no conception of the myriad of costs (both fiscal and performance related) associated with cloud computing.

Cloud computing has enabled even small organisations to deploy thousands of VMs, if only for short period of times. Prior to cloud computing this level of scale was unavailable to all but the largest organisations, as such the vast majority of legacy monitoring software is designed for smaller scale operations. As the scale of cloud deployments continues to grow the availability of scalable monitoring tools which support the unique requirements of cloud computing are becoming necessary. Monitoring as a service tools are often touted as those tools, however these services lack the customisability of previous tools and lack any mechanisms to implement corrective or adaptive behaviours. Any tool which is designed to monitor large deployments of virtual machines must be autonomic. It cannot rely upon humans to process events or analyse data and implement manual alterations. Humans are simply too slow to manually monitor anything resembling a large scale system. What is therefore required are tools which can collect and analyse monitoring data and decide if necessary to alter the state of the system. This is significantly beyond the current state of cloud monitoring and thus requires new tooling.

The area of cloud monitoring remains relatively unexplored and there is hitherto no universally accepted toolchain or systems for the purpose. Most real world cloud monitoring deployments are a patchwork of various data collection, analysis, reporting, automation and decision making software. There are few universal best known practices or tools and many grid and cluster monitoring tools remain in common usage despite being a poor fit for cloud monitoring.

This paper presents a detailed overview and evaluation of Varanus [2–4], 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.

Prior work

Cloud monitoring is a relatively new area, however other types of monitoring have produced a vast array of tools and designs. Many of these tools have not been designed with cloud properties in mind and are potentially ill suited for cloud monitoring, irregardless these tools are frequently used and referred to within the domain of cloud monitoring.

Nagios

Nagios [5] is the premier open source monitoring tool. Initially released in 1999, the venerable tool has a vast plugin library which supports virtually all common software, network devices and appliances. Nagios was never designed for cloud computing, as such it has no native support for tolerating elasticity and requires a significant deal of modification to better support cloud computing. Despite the domain mismatch, Nagios remains the most popular monitoring tool for two reasons: its extensive functionality and the lack of a suitable replacement. Recent cloud monitoring tools predominantly fall under the categorisation of monitoring as a service tools which eschew the computational costs of data collection and analysis to a third party company. Users who wish to keep their monitoring tools behind their firewall, or who demand greater functionality that is offered by monitoring as a service tools have little alternative to Nagios and its contemporaries. As it is open source, widely used and is available for experimentation we therefore make use of Nagios to compare with Varanus throughout our evaluation described in detail in Section 8. Two versions of Nagios are evaluated alongside Varanus: a stock configuration and a modified installation which has a number of patches applied including the 'large installation tweaks'. The Nagios architecture that we employ in our evaluation is depicted in Fig. 1. We employ a three tier Nagios hierarchy whereby the master is responsible for collating and analysing the results which are obtained from monitored hosts via a set of Nagios slaves. The slaves are intended to alleviate the burden of scheduling and communicating with individual VMs. Puppet is also used in order to seed newly joining VMs with the Nagios client and provide the configuration necessary to communicate with the Nagios master and additionally to restart the Nagios master in order add the new hosts.

Ganglia

Ganglia [6] is a resource monitoring tool primarily intended for HPC environments. Ganglia organises machines into clusters and grids. A cluster is a collection of monitored servers and a grid is the collection of all clusters. A Ganglia deployment operates three components: Gmond, Gmetad and the web frontend. Gmond, the Ganglia monitoring daemon is installed on each monitored machine and collects metrics from the local machine and receives metrics over the network from the local cluster. Gmetad, the Ganglia Meta daemon polls aggregated metrics from Gmond instances and other Gmetad instances. The web frontend obtains metrics from a gmond instance and presents them to users. This architecture is used to form a tree, with Gmond instances at the leaves and Gmond instances at subsequent layers. The root of the tree is the Gmond instance which supplies state to the web

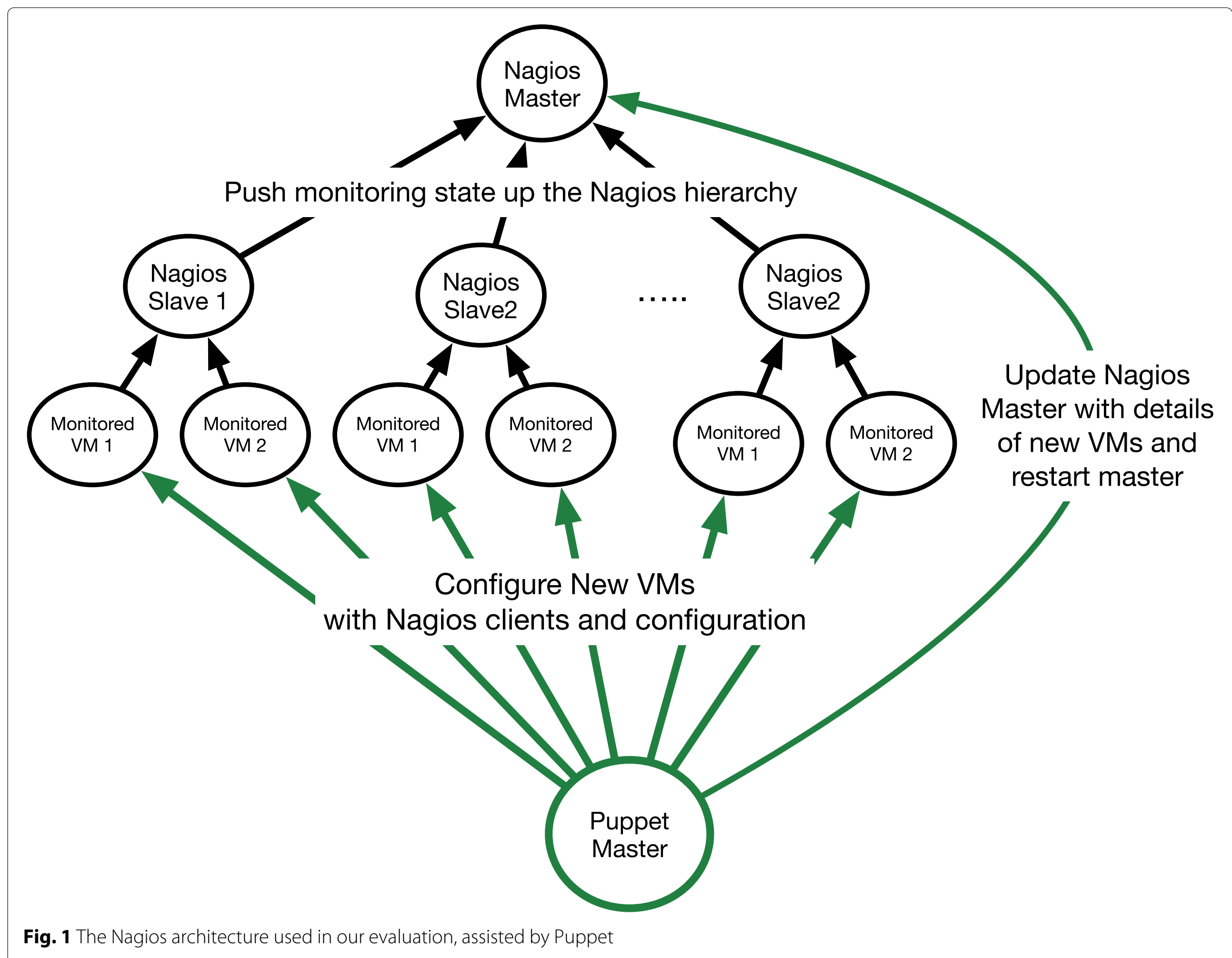

Fig. 1 The Nagios architecture used in our evaluation, assisted by Puppet

frontend. Ganglia is first and foremost a resource monitor and was designed to monitor HPC environments. As such it is designed to obtain low level metrics including CPU, memory, disk and IO. It was not designed to monitor applications or services and nor was it designed for highly dynamic environments.

Riemann

Riemann [7] is an event based distributed systems monitoring tool. Riemann does not focus on data collection, but rather on event submission and processing. Events are representations of arbitrary metrics which are generatedq by clients and encoded using Google Protocol Buffers [8] and additionally contains various metadata (hostname, service name, time, ttl, etc). On receiving an event Riemann processes it through a stream. Users can write stream functions in a Clojure based DSL to operate on streams. Stream functions can handle events, merge streams, split streams and perform various other operations. Through stream processing Riemann can check thresholds, detect anomalous behaviour, raise alerts and perform other common monitoring use cases. Designed to handle thousands of events per second, Riemann is intended to operate at scale.

Amazon CloudWatch

CloudWatch [9] is the monitoring component of Amazon Web Services. CloudWatch primarily acts as a store for monitoring data, allowing EC2 instances and other AWS services to push state to it via an HTTP API. Using this data a user can view plots, trends, statistics and various other representations via the AWS management console. This information can then be used to create alarms which trigger user alerts or autoscale deployments. Monitoring state can also be pulled by third party applications for analysis or long term storage. Various tools including Nagios have support for obtaining CloudWatch metrics.

Access to this service is governed by a pricing model that charges for metrics, alarms, API requests and monitoring frequency. The most significant basic charge is for metrics to be collected at minute intervals followed by the charge for the use of non standard metrics. CloudWatch

presents a trade-off between full customisability and ease of use. The primary use case of CloudWatch is monitoring the full gamut of AWS services. Users of only EC2 will likely find the customisability of a full monitoring system preferable to the limited control afforded by CloudWatch.

Varanus overview

Varanus is comprised of four components: the coordination service, the collection service, the storage service and the analysis service. The next four sections detail the design of these services. Figures 2 and 3 provide a high level overview of the interaction of the four services. These loosely coupled services cooperate in order to provide a full suite of monitoring functionality. The coordination service is the foundation upon which the other services operate, providing a means for the components to communicate and provides VM registration, configuration storage, decision making and agreement. The data collection service is comprised of a small daemon which operates on each monitored host which collects metrics and values and transmits them to the storage service. The storage service runs across elected (or specifically dedicated) VMs and provides a mechanism for the in memory storage and processing of large volumes of timeseries data. The analysis service consumes data from the storage service in order to detect irregularities, failure, bottlenecks, plan optimisations and other user definable behaviour.

Varanus coordination service

The coordination service is a robust, highly available configuration store which additionally provides agreement, configuration storage and failure detection to the other components within Varanus. It is a self contained service with no external dependencies and is intended to continue operating even under high failure rates. Every VM within a Varanus deployment runs a coordinator daemon, however the role that each coordinator takes can vary from taking part in relevant agreements and detecting failure to storing configuration data, coordinating agreements and enforcing consistency.

When building distributed systems the challenge of configuring and coordinating components soon arises. In the typical case one looks to well tested frameworks and platforms upon which to build. Software such as Apache Zookeeper, etcd and Doozerd provides fault tolerant mechanisms for service discovery, coordination and orchestration. The use of existing configuration and management tools as a basis upon which to develop a monitoring tool poses a number of issues. Firstly, such tools are commonly built upon large stacks which require a myriad of dependencies resulting in a footprint far beyond what is required. Ideally, in order to avoid a significant observer effect the components of a monitoring tool must be small and unobtrusive. Secondly, current coordination tools are intended to be used by multiple applications. Thus, the failure, loss of performance or other issues with a coordination service would impact both critical applications and the monitoring service. Therefore, when the monitoring tool is most required, it is unavailable or degraded. We therefore eschew the use of third party coordination tools in favour of a dedicated out of band mechanism which is intended to tolerate a wide variety of failure modes in order to facilitate monitoring and maintain a small footprint. For this purpose we look towards peer to peer overlay networks as a means to provide a highly robust basis for developing a monitoring solution.

Cloud computing is unlike classical peer to peer computing scenarios whereby there are a large number of geographically distributed peers, each with network conditions of variable performance. A typical large scale cloud deployment consists of a significant number of VMs

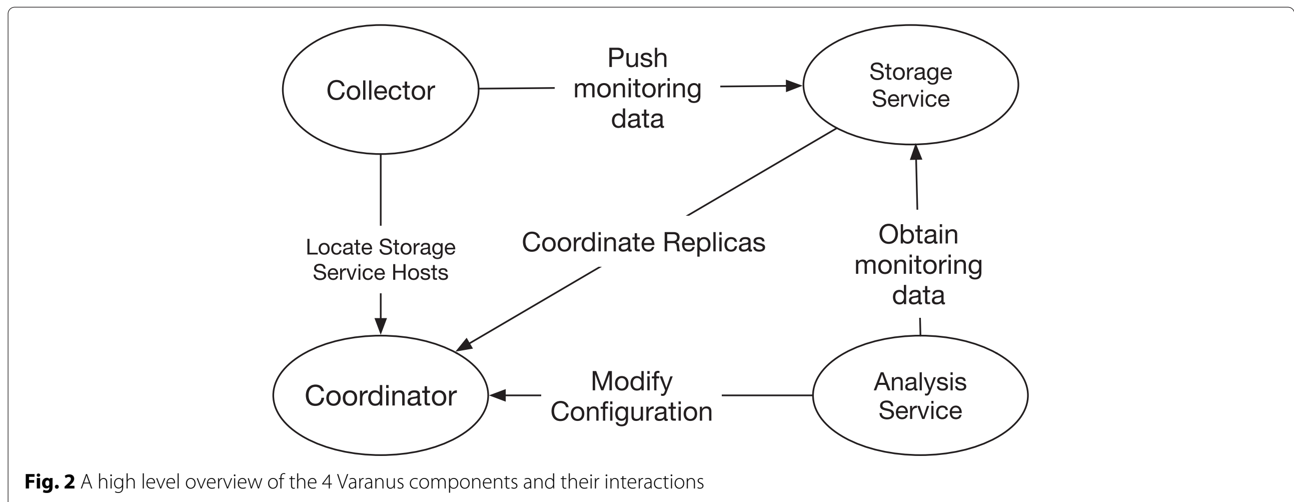

Fig. 2 A high level overview of the 4 Varanus components and their interactions

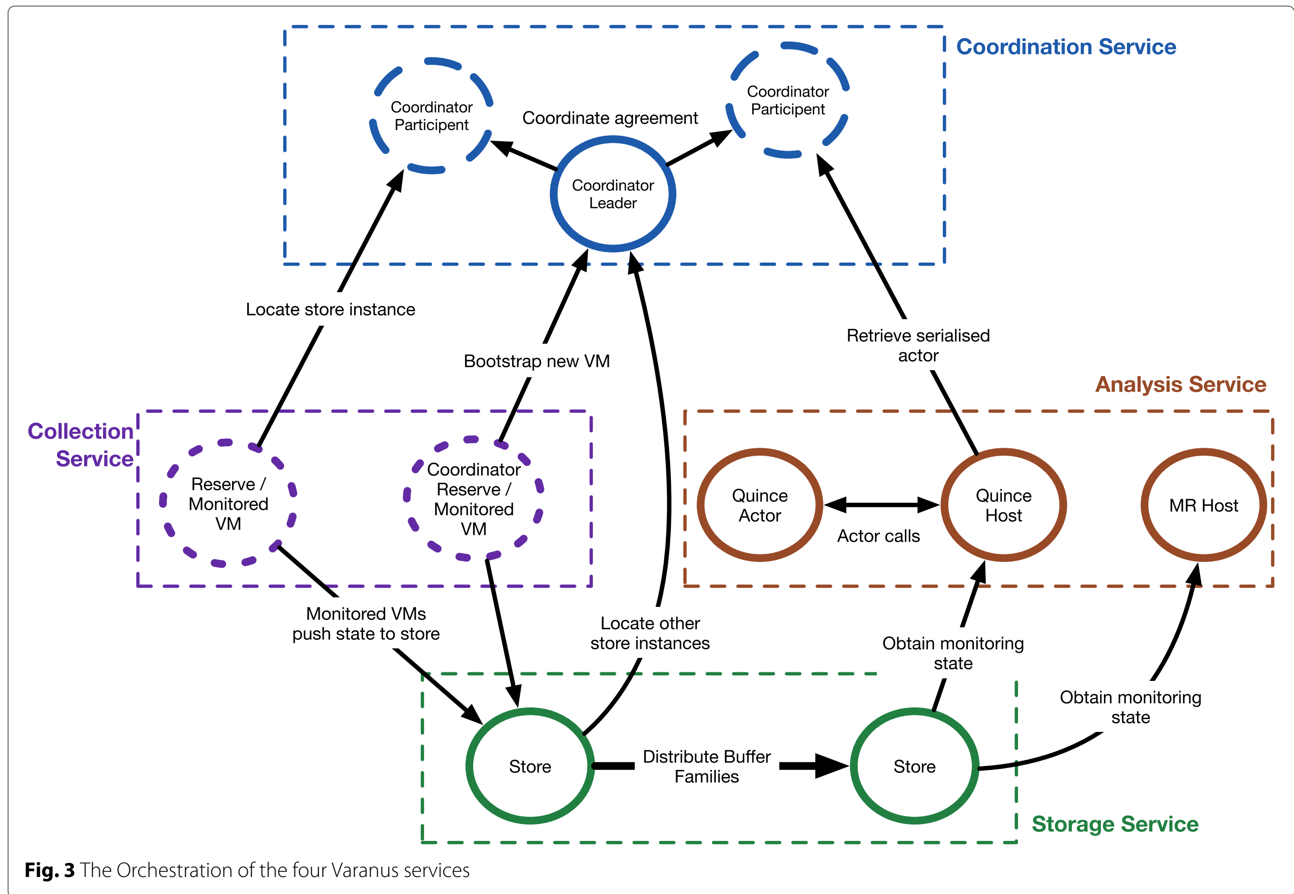

Fig. 3 The Orchestration of the four Varanus services

in a small number of geographical locations with each (in usual circumstances) possessing plentiful bandwidth within the localised cloud *region* and reduced bandwidth between cloud regions. This scenario lends itself to the use of a structured peer to peer architecture which exploits localisation and the plentiful bandwidth within cloud *regions* while conserving slower inter-cloud bandwidth. The Varanus coordination service is peer to peer overlay network which attempts to exploit the architecture of clouds.

We describe the communication architecture employed by the coordination service in terms of three groupings: cloud, *region* and sub-region. Cloud and *region* map neatly to the well established terminology, a cloud is a top level abstraction which includes multiple *regions* and a *region* is a geographic area which hosts cloud resources. A sub-region is a further sub division of a *region* which includes a subset of provisioned cloud resources this is similar to the notion of (availability) zones but may transcend or overlap actual zones.

These different levels of abstraction produce a three tier hierarchy. An example of this hierarchy is shown in Fig. 4 and a more detailed overview is provided by Fig. 5. The coordination service employs a gossip protocol over this hierarchy in order to facilitate a range of functionality. The coordinator itself uses the protocol to disseminate configuration state, update membership and detect failure meanwhile the other service uses this mechanism to exchange state.

Communication

The coordination service makes use of a gossip protocol. In the context of cloud monitoring, gossip protocols have several advantages, including:

- Minimising CPU usage in favour of utilising network capacity. This is advantageous in a cloud computing setting where internal bandwidth is free where as CPU is a metered resources.
- Tolerating network outages, membership change and failure with minimum overhead.
- Providing heartbeat and primitive failure detection at no extra message of computation cost.

Therefore, a gossip protocol serves as the communication mechanism for the coordinator and for the other Varanus services.

In large scale cloud deployments individual VMs operate under a range of computation and communication

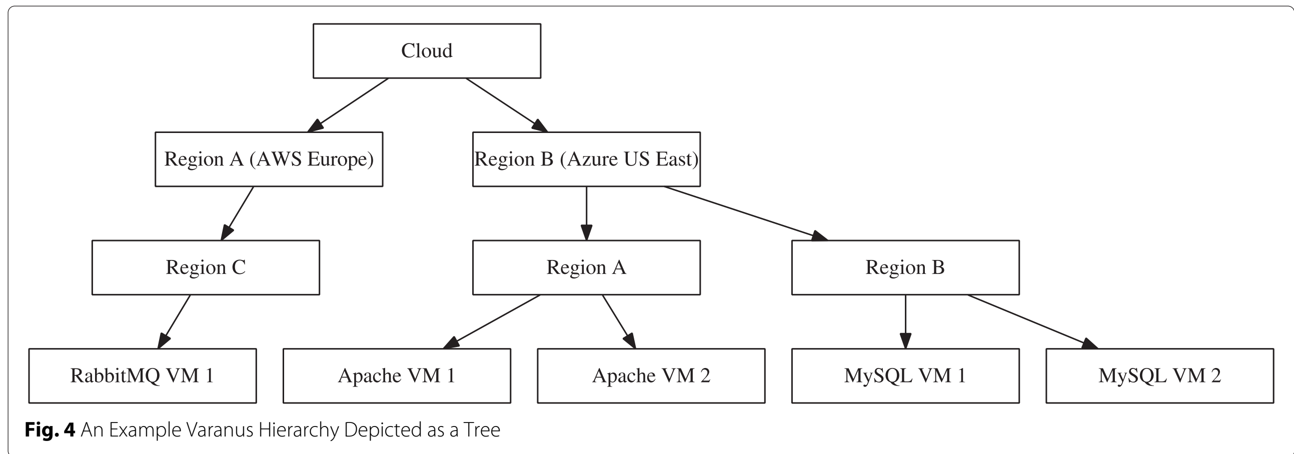

Fig. 4 An Example Varanus Hierarchy Depicted as a Tree

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 been demonstrated to be effective mechanisms for providing robust and scalable services for distributed systems including information dissemination [10], aggregation [11] and failure detection [12]. The coordination services uses a gossip protocol to propagate updates to the configuration store, update membership and detect failure. In addition to the gossip protocol, the coordinator also makes use of the Raft protocol in order to achieve consensus when necessary.

The basic operation of the coordination gossip protocol consists of the periodic, pairwise propagation of state between coordinator instances. 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:

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

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 timestep. 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 correspondents selected as targets).

In addition to this mechanism, preferential target selection is used to reduce the delay in propagating state. Targets are selected based on a weighting scheme which uses round-trip time estimates in order to select targets which are topologically closer. Each round of gossip is spatially weighted according to the scheme proposed in [13], using RTT as a distance metric in order to propagate

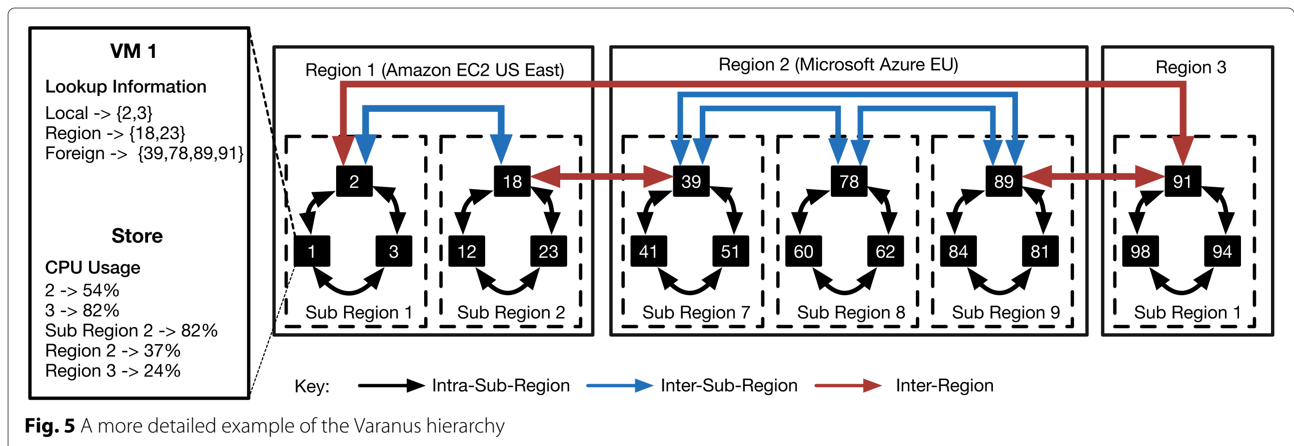

Fig. 5 A more detailed example of the Varanus hierarchy

updates to all VMs 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 churn and failure.

In order to best exploit the topology of IaaS clouds different behaviours occur 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.

This gossip protocol, 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.

1. Intra Group: communication between monitoring agents within the same sub-region. This occurs at a near constant rate. Each time an event occurs in the coordinator or other Varanus service the coordinator propagates the new state to its 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 *sub-regions* 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 and a small subset of local contacts and foreign contacts are propagated.
3. Inter-Region: communication between coordinator processes in different different cloud *regions* or datacenters. This occurs proportionally to the inter-group rate. At this level aggregate values for the entire region and subsets of the local and foreign contacts are propagated between regions.

Some concrete examples of this communication hierarchy include:

- Configuration store lookup data. A full set of lookup data is sent to hosts within the local sub-region, the location of top level VMs are sent to *sub-regions* within the same region and only the location of the root VM is propagated between regions.
- Monitoring data collected by the collection service. Raw data is sent to storage service instances in the same sub-region, aggregates values of the *sub-regions* resource are sent to neighbouring *regions* and aggregates of the entire *region* are sent to other regions.
- Membership information. Full membership information is propagated within a sub-region. Between *regions* a subset of hosts in that *sub-region* are propagated and between *regions* a small sub-section of hosts in the *region* are propagated.

Consensus

In addition to the gossip protocol, the coordination services makes use of a separate protocol for enforcing consensus. This protocol is less frequently used than the gossip protocol due to its additional complexity and overhead. In the coordination service it is used for leader election, consistent commits and the other services rely upon it for a range of functions. The Varanus consensus protocol is based on the Raft consensus algorithm [14]. Raft is used as it is comparable to Paxos in terms of performance, but it more modern and is designed to be easier to understand and debug in addition to having a wide range of implementations. Raft uses a replicated state machine approach to consensus similar to Paxos but is intended to be simpler to understand and to implement. In order to agree upon a single value, Raft employ a leader to enforce consensus. The process of committing a value using Raft is as follows:

1. A leader is elected from the pool of candidates.
2. The leader continuously broadcasts heartbeats to followers. Follower use a 200–500 millisecond heartbeat timeout which varies based on acknowledgement time.
3. Followers respond to the heartbeat with an acknowledgement.
4. A client submits a value to the leader.
5. The leader attaches the value to the heartbeat and waits for acknowledgements.
6. Once the majority of VMs have accepted the value the leader commits the value.
7. The leader then appends a notification to the heartbeat to notify all followers of the agreed upon value.

Should followers fail to receive a heartbeat it will become a candidate, nominate itself and broadcast a solicitation for votes to all other VMs. Should a network partition occur, the leader of the partition with the largest portion of VMs will be able to commit. Other partitions will not. After the partition ends, values are reconciled to return all VMs to a consistent state.

Each stage where the leader or a candidate broadcasts to all followers is performed using the Varanus gossip protocol. Acknowledgements are more standard unicast

messages. Batches of commits and acknowledgements can be compacted into single messages which reduces the complexity cost of performing frequent commits. Despite this, a single commit requires several rounds of gossip to complete and as such is avoided in favour of plain gossip wherever possible.

sub-region assignment

sub-regions are groups of related VMs within the same geographic cloud region. Related is qualified by a number of factors including the software the VM is running, the network distance and latency between VMs and the behaviour of the VM.

sub-region assignment is done according to a distributed, weighted, k-nearest neighbour algorithm. Upon instantiation the collection service daemon running on each VM compute a feature vector which describes all available properties including installed software, resource usage, logical location and user provided metadata. This vector is then pushed to the coordination service. A default weighting is given in favour of installed software and logical location (location in terms of cloud *region* and in terms of network distance). This weighting is given as the software the VM is running is the most likely factor in determining the purpose and general behaviour of the VM and location has the greatest influence on the cost of communication. This scheme therefore preferentially groups VMs running similar software which have few network hops between them. The feature vector 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.

The coordination service computes an aggregated feature vector for each preexisting *sub-region* describing properties common to all VMs within that sub-region. Newly instantiated VMs fetch all relevant group's aggregate feature vector from the coordination service and perform a k-nearest neighbour to assign the VM to a *sub-regions*. Should the distance between the VM and existing sub-groups exceed an acceptable value or should no *sub-regions* exist, the VM will form a new sub-region. Periodically the coordination service recomputes the feature vector for each *sub-region* to ensure it best reflects its membership. Should a the individual feature vector of a VM differ significantly from its sub-region aggregate it will perform a comparison against other relevant *sub-regions*. After a delay, if no satisfactory *sub-region* can be found, he VM will depart its *sub-region* to form a new sub-region. If a *sub-region* remains underpopulated, when compared to other *sub-regions*, its members will disband and join the other *sub-regions*. After a repeat of this process, VMs will cease forming new *sub-regions* for an exponentially increasing length of time in order to prevent an infinite cycle occurring. This grouping scheme attempts to group related, nearby VMs based upon the assumption that monitoring state is most valuable to VMs similar to that from which the state is collected. Varanus, which is based upon this scheme, is therefore primarily concerned with the distribution of monitoring state to other VMs and places delivering state to human users as a secondary concern. This is pursuant to Varanus being an autonomic monitoring framewo *sub-region* rk and is motivated by the ability of software to make effective use of large quantities of near real time monitoring state (as opposed to humans' lesser capacity).

Membership

Keeping track of membership of a large scale system requires significant message rates. A number of schemes have been proposed in peer-to-peer literature [15–18] which provide mechanisms for disseminating and maintaining membership state at each peer. Varanus, however, aims to be unobtrusive and have limited effect upon monitored VMs. As such, the coordination service makes no effort to maintain a consistent global view, or anything approaching a global view. Rather, the coordination service local to each *sub-region* tasks itself with tracking full membership of that *sub-region* and maintains two additional member sets storing minimal membership state of *sub-regions* belonging to the local region and remote *regions* respectively. These sets are referred to as local contacts and remote contacts.

This scheme provides a means for the coordination service to locate related VMs quickly, only requiring them to consult their own state store. Meanwhile, should the need arise to communicate with monitoring agents in other *sub-regions*, lookup can be achieved in constant time via the local and remote *sub-region* contacts. This scheme also allows a global view to be built, if necessary, with relative ease.

Newly joining VMs register their membership by gossiping with the coordination service in that sub-region. This in turn will eventually announce that VMs presence to other *sub-regions*.

Role assignment

Each VM runs a coordinator instance, however not all instances perform the same functions. Coordinator instances within a *sub-region* can be assigned one of three roles:

1. The leader is responsible for committing values to the mutable configuration store and ensuring its consistency. Similarly the leader enforces agreement amongst its sub-region. It also serves as the broker for events and holds a portion of the configuration store. It is elected using the consensus mechanism.
2. Failovers are participants which serve as hot standbys should the leader fail or elect to stand down.
3. Participants store portions of the configuration store and take part in agreement.
4. Reserves receive event notifications and other message and are able to become participants but take no active role in the coordination service until they do.

The leader is assigned using the standard agreement mechanism from the pool of participants. The election considers load average as the worthwhile value, the initially elected VM is the VM with the lowest load average. Participants also nominate themselves based upon load averages. If a VM is heavily loaded it can opt to become a reserve and take on no additional work that could affect its performance. If the leader or a participant encounters a sustained period of load it can opt to become a reserve. In this case all Varanus services other than the collection service must migrate their state to alternative hosts (unless already done so). If all available hosts are acting are reserves, Varanus cannot operate correctly. In this situation Varanus require additional dedicated VMs in order to provide monitoring functionality.

Regions and clouds also have leaders. *region* leaders are nominated from a pool of *sub-region* leaders and cloud leaders are nominated from a pool of *region* leaders. This allows configuration storage, agreement and other functions to be performed across *sub-regions* is necessary. The mechanisms which govern how *region* and cloud leaders operate is the same as *sub-region* leaders.

Configuration store

While Varanus attempts to be autonomic, providing monitoring services with as little human interaction as necessary, it still has a need for configuration data. Archetypal design advises the use of configuration files that use a standard format such as XML, YAML, .cfg or .ini. This design requires either a human or a program to write the file and raises a number of potential issues regarding file versioning and consistency. While this design can be successfully used (we employ this design in the collection service in order to integrate existing software) it is best to avoid it entirely and employ a design which lends itself to programatic (and thus autonomic) configuration. This is the role of the configuration store.

This portion of the coordination service lends itself to direct comparison with Zookeeper, etcd and other services. Unlike these other tools the Varanus coordination service is fully decentralised and designed specifically to support a single set of services. This allows the configuration store to be vastly simpler than other general purpose services and potentially more robust.

The configuration store is comprised of several individual stores. Each level of the communication hierarchy has its own configuration store. There is:

1. A cloud wide store which stores values relevant to every VM in the cloud.
2. A *region* wide store which stores values relevant to every VM in the region.
3. A *sub-region* wide store which stores values relevant to every VM in the sub-region.

Each store is managed by the leader of the respective level in the hierarchy. Leaders pass commits down the hierarchy until the value is committed to participants within a sub-region. Thus, a value committed to the cloud store will be replicated in every sub-region, a value committed to the *region* will replicated in *sub-regions* belonging to that *region* and the *sub-region* store has no replication beyond that sub-region.

Each store both in the hierarchy has an independent keyspace. The keyspace of each store is hierarchical and resembles a UNIX filesystem as shown in Fig. 6. The store supports three types of node: directory nodes, data nodes and immutable data nodes. Each keyspace has a top level root node which subsequent nodes are created under. Data nodes can either be top level or can be children to directory nodes. Directory nodes can also have directory nodes as children. Directory nodes provide two operations: get and delete. Get returns a list of known children and delete removes the node and any children. Mutable data nodes provide four operations to clients: get, update, delete and watch. Get returns the value, update updates the value of the node, delete removes the node and watch registers a client's interest in that node such that on a value change an event is raised and sent to the client. Immutable data nodes are regular data nodes which only support the get operation.

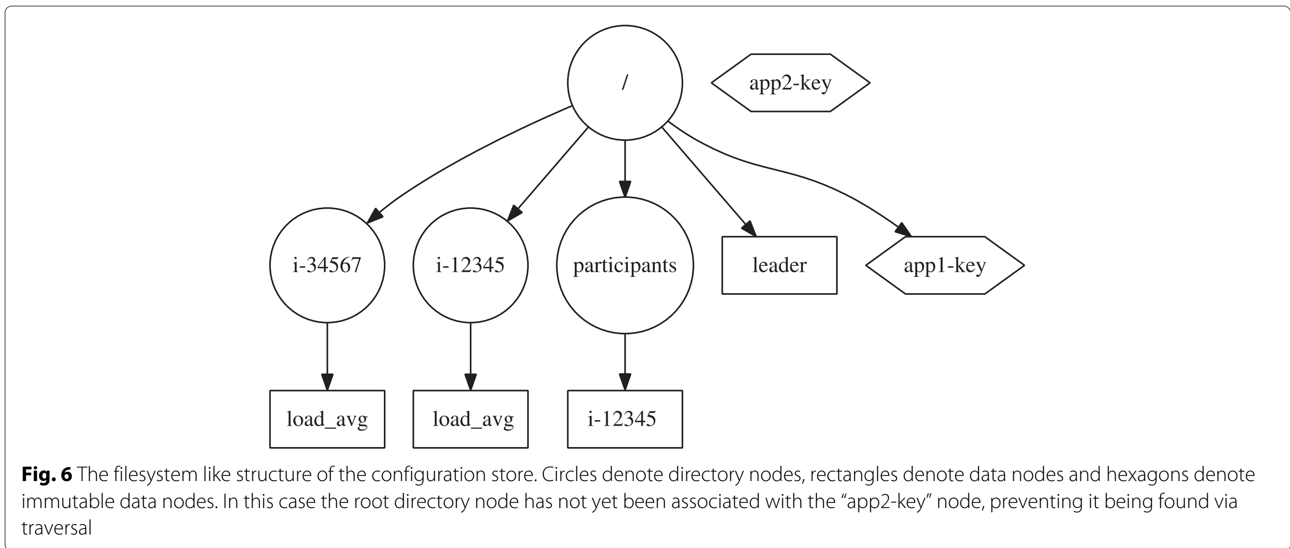

Fig. 6 The filesystem like structure of the configuration store. Circles denote directory nodes, rectangles denote data nodes and hexagons denote immutable data nodes. In this case the root directory node has not yet been associated with the "app2-key" node, preventing it being found via traversal

Consistency is the motivation behind the dichotomy between mutable and immutable data nodes. As the immutable nodes do not support modifying existing values there is no need to employ any complex consistency protocol to ensure that all copies of that value are kept consistent. Adding immutable nodes to the store can be performed by any participant (where as mutable values can only be committed via the leader) and can make use the the previously described gossip protocol to quickly disseminate state to other participants. The mutable data nodes and directories meanwhile, allow deletion and allow values to be updated and to avoid inconsistent configuration requires the use of an consistency protocol. For mutable values the store makes use of the Raft based consensus mechanism. As per all consistency algorithms, RAFT is significantly slower than the Varanus gossip algorithm.

Immutable nodes can be created without modifying directory nodes. If this is done, the list of children provided by the parent will not include the immutable node. This is acceptable if the immutable node's key is known, otherwise this is problematic as clients will be unable to locate the node. This can be rectified by updating the directory node which, while more costly than creating the immutable node, is less costly than creating a mutable node. Updates to the directory node can be done in batches to allow numerous immutable nodes to be created before the directory is updated in a single operation.

Both the mutable and immutable nodes use the same strategy for determining replica placement. Replication is orientated around *sub-regions*. Three factors are used in determining replication locations:

1. Load average, highly loaded hosts are avoided while underloaded hosts are preferred
2. Uptime. All VMs are eventually terminated and often termination occurs in batches. It is therefore advantageous to distribute keys over hosts which have a range of up times so as to potentially avoid all replicas being terminated simultaneously.
3. Previous keys. So to avoid a overly skewed distribution of keys the number of current keys stored by each host is considered and hosts with fewer keys are preferred as replicas.

Priority is given to load average, pursuant in the goal of Varanus being unobtrusive. The coordination service will attempt to identify K replicas (where K is a user defined value stored by the configuration service, defaulting to 3) within a *sub-region* which are uniformly distributed throughout the range of up times which have not been overloaded and have fewer than half of the keys assigned to the most significant replica. If the coordination service cannot find K replicas which satisfy these criteria it will relax the need for a uniform distribution and accept replicas with more than half the number of keys that the most significant replica stores. If it still cannot satisfy those requirements it will relax them further, until a suitable arrangement can be found.

Each coordinator maintains a lookup table for values within each store. Each participant within a *sub-region* periodically gossips a subset of keys which it is currently responsible for. Keys are chosen uniformly at random. Every other node in that *sub-region* receives those messages and updates its lookup table. A timeout, proportional to the average time between updates and the number of keys in the *sub-region* is used to remove stale entries.

The configuration store is intended to store small (less than 4KB) items. The store is intended to store configuration strings and data necessary for the coordination service to function. Larger data sets, such as full monitoring data is committed to the storage service which is optimised for the storage of greater volumes of time series information. The information that the configuration store is intended to store includes:

- 15 minutely load averages for members. This is used for leader elections and for replicas.
- Configuration strings, for example the key-value pair: "n_replicas=3".
- The location of the leader and participants

Failure detection

Failure detection is provided by a gossip based Phi Accrual algorithm [19], using a similar scheme to Apache Cassandra. Heartbeats are propagated between coordinator processes via a gossip protocol at a regular interval. Rather than providing a boolean failed/alive value, the phi accrual failure detectors provides Φ, a value signifying the likelihood that a given VM has failed. Φ is derived from a sliding window of heartbeat intervals. The failure detector calculates the mean, median and variance within the window and builds the resultant exponential distribution which is used to compute Φ. This differs from the original Phi Accrual design which used the normal distribution as the exponential distribution better tolerates the variable latency of the gossip stream. This algorithm is beneficial for cloud environments whereby there is often considerable variance in network performance. If the network performance degrades the resulting averages and variance will increase resulting in a longer period than previously for the value of Φ to rise. When the Φ of a given VM exceeds the a predefined threshold that VM is considered to have failed by the detecting VM. The detecting VM first checks using the cloud provider's API if the failed VM has been terminated, if it has indeed been terminated it announces this to other VMs and no further action is taken. If, however, the VM is still declared as running by the cloud provider then the detection VM initiates a round of consensus in order to agree upon the failure. If consensus is reached, the VM is declared failed and action can be taken to mitigate that failure. If agreement cannot be reached it is therefore the case that some VMs are not receiving heartbeats while others are. This indicates the presence of some form of network partition or other Byzantine failure. One of the strengths of the RAFT consensus algorithm is the inbuilt ability to detect network partitions and still operate in spite of them. Thus, if a round of agreement is initiated and there is indeed a network partition, so long as their is a majority of VM in one partition, the VMs that can still communicate with each other can still reach consensus regarding the failure. The default action for failed VM (including VM that are divided by partition) is to terminate them and replace them with correctly functioning VMs.

For optimum failure detection, every VM should subscribe to the heartbeats of every other VM and calculate the appropriate Φ value. This is ill advised for a number of reasons: firstly it would result in significant computation at each VIM and secondly would require heartbeats to be propagate across cloud regions. Heartbeats, are instead propagated according to the previously described gossip hierarchy. The full set of heartbeats are propagated within sub-regions, a subset are propagated within regions and a smaller subset between regions. It is subsections, not aggregates that are transmitted at higher levels of the hierarchy as aggregates would be of little value and would not be appropriate for the Phi Accrual algorithm. Subsections are still, however, propagated at a decreasing rate as accorded by the hierarchy. The Phi Accrual algorithm perfectly tolerates slow but constant heartbeats and as such is unaffected by a slowed rate of propagation at higher levels of the hierarchy. The size of the subset that is propagated between sub-regions and regions is dependant upon predefined configuration. Propagating heartbeats between sub-regions doesn't particularly aid in failure detection, as gossip messages are propagated faster within sub-regions inevitably failure will be detected there first. What propagated gossip messages does achieve, however is determining if a network partition has occurred between layers of the hierarchy. Unlike in the case of failure detection with a sub-region, terminating VMs is unlikely to resolve this issues but detection can allow alternative action to be taken.

Events

Coordinators and other Varanus services may have prolonged interest in values stored in the configuration store or may need to take special action should a VM fail or other phenomena to occur. One approach to this problem is to continuously poll the values or VMs in question and take action should the result of that polling change. This approach consumes unnecessary cpu cycles and bandwidth and risks becoming intrusive. In lieu of this approach Varanus employs an asynchronous event system which can be used to alert watchers as to value changes as serve as the basis for an event based programming model.

Varanus events are immutable associative arrays which contain a number of fields which describe the event in detail. The Varanus event formats is shown in Fig. 7. Events can either serve as a notification, such as in the case of watchers whereby the arrival of an event triggers a callback, or a means to encode state and invoke remote computation. Actors can be propagated via plain unicast or using the gossip protocol. Events are discussed

Field	Purpose
Host	the hostname or IP address that the event pertains to
Origin	the source of the event if different that the host it pertains to
Subject hline Id	the subject of the event, used for pub-sub event distribution. Subscribers subscribe to subjects. a unique identifier for the event
Service	the service or application that the event pertains to e.g. Apache, System, MySQl etc
Type	the type of event, used to specify actors
Value	the value relevant to the event
Timestamp	the time the event was produced, used in conjunction with the Id to determine the lifespan of messages
Tags	(Optional) an unordered list of key-value pairs for providing additional data
Preprocessor	(Optional) the name of a Quince actor to invoke before handling
Arguments	(Optional) an ordered lists of parameters for the preprocessor

Fig. 7 The Varanus event format

with regards to computation and analysis in section 1. Events are distributed over the existing coordination service infrastructure and therefore serve as a lightweight loosely coupled message service, they are used by the analysis service to encode intermediate computation and by other services to notify components of configuration and status changes.

Varanus collection service

Monitoring fundamentally entails the observation of behaviours which occur within a system. Most data collection tools, the Varanus collection service included, consist of a small daemon which regularly collects state and transmits it to a remote host for storage and processing. What differs about the Varanus collection service is its ability to perform on the fly reconfiguration and alter what data is obtained and the frequency and granularity at which it is captured.

A single VM has no shortage of values and behaviours which can be monitored. In the typical use case interest focuses around a limited number of values, these are primarily performance factors (cpu, memory, transactions, requests etc), application behaviours (error rates, logs, status codes etc) and user behaviour (click through rates, time per page etc). There are additional sources of potentially valuable monitoring data which vastly outnumber the typically collected data. Virtually all changes in memory or on disk and all network activity can be valuable. Be it for diagnostic purposes, intrusion detection or another more specific motivation. Capturing the full range of potential metrics is, however, all but impossible. The computation required to collect all possible metrics would outstrip all other applications and much of the data may be of no practical use for a given use case. Therefore best practice is to collect a subset of the possible metrics which are most appropriate to the monitoring system's use cases. A monitoring system has a wide range of use cases and the importance of a given use case may change dependant upon the state of the system. For example if a system is suffering from widespread failure which is preventing user's accessing a service, diagnostic metrics increase in importance while use metrics decrease. Situations may also arise where it is advantageous to have data collected at a shorter interval or at a different level of precision. It is therefore important to provide a flexible mechanism to collect monitoring state that befits the current set of use cases. The Varanus collection service aims to be such a mechanism. The collection service considers three types of monitoring data that can be obtained:

- Metrics, numeric values typically representing performance
- Logs, debugging data, service codes and other textual data which describes events or behaviours

Just as there is no shortage of state to collect, there is no shortage of tools to perform the collection. Nagios, Ganglia, Zenoss, Amazon CloudWatch, DataDog, New Relic and many more tools replicate near identical functionality which obtains metrics from a host. Many of these tools rely upon the same low level APIs and tools to obtain said data and given the relative lack of options to obtain certain types of data, any attempt to design a new tool would inevitably repeat the functionality of past tools. For example, the venerable Apache HTTP server provides a handful of ways to obtain its internal state, developers are limited to parsing log files or parsing the output of a status page (provided by mod_status). Furthermore, the strength of many well adopted data collection tools are the plugin architectures and vast plugin libraries that enable existing tools to obtain data from new applications. It would therefore be foolhardy to eschew existing data collection tools however existing tools are not without their shortcomings. The vast majority of current tools rely upon manual configuration and are not interoperable by default. Furthermore many data collection tools can be discounted due to their plugin architectures being little more than

a myriad of shell and Perl scripts which are difficult to maintain and autonomically deploy.

From the vast pool of potential tools, the Varanus collection service uses collectd, a well established data collection daemon with an extensive plugin library, to obtain data from the OS and applications. collectd is an ideal basis for building a more complex data collection tool as it has a small footprint (in terms of resource usage and its size on disk), has a wide range of plugins, an active development community and a simple, well designed architecture. The design of collectd is orientated entirely around plugins. The core of collectd does very little, its primary function it to invoke plugins which obtain data and then to invoke plugins which stores or transmits that data. Exactly which plugins are invoked, what data is collected and where it is transmitted is determined by a configuration file. This limits the effectiveness of collectd as an autonomic tool and necessitates a human or an external service provide this configuration. In large deployments it is common place to use Chef, Puppet, Salt or an alternative automation tool to install and configure collectd instances. For deploying and managing a monitoring tool, it is undesirable to have external dependencies as it is most desirable to ensure that the monitoring tool continues to function despite the failure of external services. It is for this reason that the collection service manages the configuration of collectd with no dependency other than the Varanus coordination service.

The collection service consists of two components: a collectd instance and a configuration generator. Additionally collectd uses a bespoke plugin which allows collectd to communicate data to the storage service. Collectd uses a unicast protocol to communicate directly with the appropriate storage service instance. The architecture of the collection service is depicted in Fig. 8.

The configuration generator is intended to solve two problems: generating the initial configuration for collectd which is appropriate for the software and services operated by the VM and secondly updating configuration to alter data collection as requirements change. In order to initially generate the necessary configuration for collectd, the config generator pulls down the set list of metrics and rate of collection from the configuration store. The configuration generator then outputs the necessary configuration in the collectd format and begins the collection process. The coordinator instance running on the VM passes any relevant events, such as updates to collectd configuration to the configuration generator. This allows Varanus to alter the rate of collection and change what metrics are collected.

The storage service

The industry standard tool for storing time series data is RRDTool [20], a circular buffer based database library. A typical RRDTool deployment will have a small number of variables stored per RRD database file. As a central responsibility of Varanus is the storage of time series monitoring data one might initially consider the use of rrdtool. There are however a number of limitations which reduce the applicability of rrdtool to Varanus and its intended use cases.

- When used as a backend for a monitoring tool an update to an rrdtool database typically occurs when fresh data is collected. An update involves multiple disk reads and writes to the header and archives within the database and if consolidation occurs there are additional reads and writes. As the number of databases increases and as the delay between updates decreases the IO load increases. This load is tolerable for hundreds of rrdtool databases however as this number increases to thousands, IO load becomes problematic. Users of medium to large scale systems report[2] [3] that IO load can result in rrdtool falling behind on writes, resulting in the database failing to represent recent monitoring data. This has lead to the development of various caching solutions which feed data to rrdtool, a Java reimplementation of rrdtool and a range of strategies to optimise rrdtool to reduce IO load.
- rrdtool does not allow updates with a timestamp earlier than the most recent updated. In a monitoring tool such as Ganglia where there is a hierarchical structure that provides an ordered stream of time series data this is not a major limitation. In alternative schemes where monitoring data is replicated between multiple locations or where data is collected according to a best effort mechanism this can be a significant issue. If, for example, network latency resulted in the out of order delivery of monitoring data those older values would be discarded. This limitation also prevents historical data from being imported, unless it is done so prior to other updates.
- rrdtool databases are created with a given time step: the interval at which data is committed to the database. Data which is committed to rrdtool during each time step is interpolated and a single value is store for the given time step. Unless data arrives at the exact time step (a virtual impossibility) an interpolated value is written to disk. Thus, any use case which requires the actual value as collected is incompatible with rrdtool. If no data is committed to the database during a time step or if an insufficient data is committed within a given time step in order to perform interpolation the update for that step contains an "UNKNOWN" value. If a data source creates data at an irregular rate and a rrdtool is configured with a large time step, data will be lost and

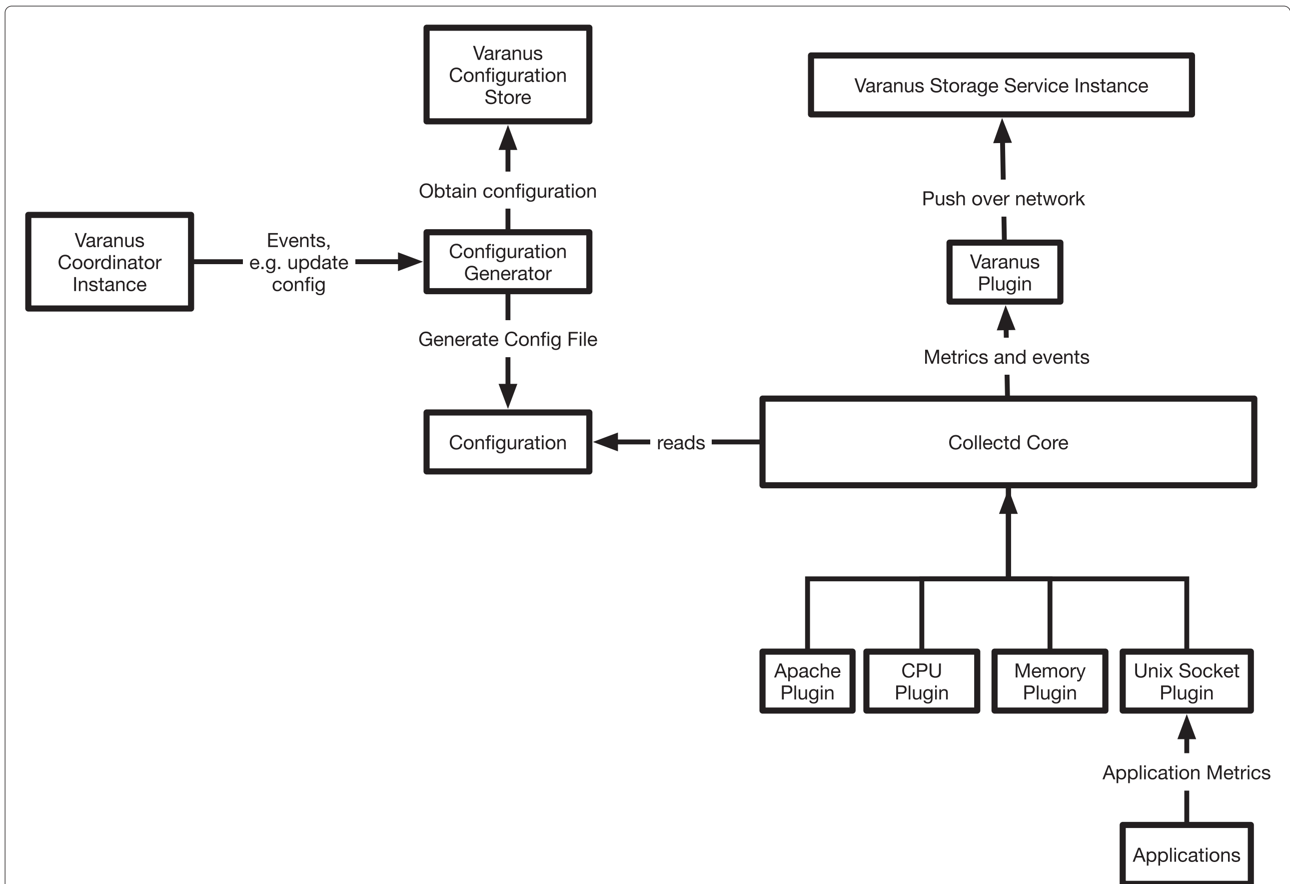

Fig. 8 The collection service. The configuration generator obtains a list of metrics and frequencies from the Varanus configuration store and generates the resulting collectd configuration. Collectd is then started and obtains state via its plugins and pushes state to the Varanus storage service through the use of a bespoke network plugin. Should a use case require additional metrics or an alternative frequency of collection then an event can be dispatched via the coordination service which will in turn be passed to the configuration generator which will update the Collectd configuration as appropriate

redundant "UNKNOWNS" will be stored in the database.

- rrdtool is first and foremost a storage format, it is not intended to support in memory process of monitoring data. All processing of monitoring data (such as threshold or trend analysis) must be done prior to or after data is committed to disk. The use case rrdstool is optimised for is graphing and it does this extremely well. Modern monitoring use cases however go beyond simple graphing and necessitate a much wider range of processing, this requires greater support for in memory storage than what stock rrdtool provides.

There are a number of plugins, patches and mitigation strategies that can make rrdtool more appropriate for cloud monitoring. While the concept of a fixed sized circular buffer database is an ideal format for storing time series data elements of the design of rrdtool, (especially the concept of a fixed update schedule and a heavy reliance upon IO) are a poor foundation for a scalable, elastic monitoring tool. We therefore propose an alternative circular buffer based storage mechanism which replicates a subsection of rrdtool functionally while making design decisions that make it more appropriate to cloud monitoring.

BufferTable

BufferTable is the Varanus storage system. BufferTable is a distributed hash table which associates a source id and metric id with a family of circular buffers. BufferTable is intended to store metrics and status codes/log/event data. As such the buffer that the family maps to varies with the format of the data.

BufferTable is an in memory data structure primarily intended to facilitate the analysis of monitoring data with each buffer family having a fixed memory footprint, persistence is achieved through a database or flat file backend. Each within a *sub-region* is eligible to run a BufferTable instance. BufferTable uses the same election mechanism

which the coordination service uses to determine the role each coordinator plays.

BufferTable instances (like all of Varanus) are orientated around *sub-regions*. BufferTable instances distribute buffer families over the VMs within a sub-region in order to store more data than a single host can accommodate. Participants are designated as BufferTable hosts if they are not fully loaded and have less than 75 % memory usage. Preference is given to hosts which have the least load and memory usage. BufferTable instances report their location via the configuration service's configuration store. A dedicated directory in the configuration store provides the locations of the BufferTable instances responsible for all collected data. The source id and metric id are used to order the directory structure. This provides the means for collectors to identify where to push state to and from where the analysis service consumes state. Collection service instances are assigned BufferTable instances to push state to on a least loaded basis.

BufferTable is indexed by a source id which contains the IP address or fully qualified domain name of the host that the metric pertains to and a metric id which identifies metric (e.g. CPUPercent, MemFree, MariaDB-Transactions etc). This facilitates constant time lookup for obtaining all metrics associated with a given host, all metrics associated with a given id and a single metric identified by a source id, metric id pair. The BufferTable does not store single circular buffers, but rather buffer families.

A buffer family is a collection of circular buffers which store time series data for a single variable. The variable can either be in the form of a floating point value or a string. The former is used for performance data, resource usage and statistics while the latter is use for representing service check results, logs, events and other non numeric data. In both cases the buffer family consists of a series of buffers appropriate to that data type. The first buffer within the family is the receiving buffer, which stores metrics as they are received from monitored hosts. Subsequent buffers in the family are aggregate buffers, these buffers store aggregated metrics representing the value over a given time period. Each aggregate buffer stores data for a given interval with each additional buffer storing aggregate for a greater time period than the previous. The user can provide intervals, via the configuration service, for aggregate buffers on a per metric basis or provide a single default set of intervals. A typical scheme might be as follows, the receiving buffer followed by aggregate buffers with a 60 s, 5 min, 10 min, 30 min and hour long periods. Values are aggregated from one buffer to another when either a buffer is full or when by a scheduled task running which runs at the frequency of the subsequent buffer. In the eventuality that a buffer has reached its capacity but the time range of data within that buffer does not extend to the interval of the subsequent buffer an interim buffer is created with an intermediate interval. This interim buffer persists until data can be directly aggregated from one buffer to the next.

In the case of the floating point buffer family aggregation is quite simple. Floating point metrics can be using a mean, median, modal, max, min, last, first or by a user defined method. In the case of string data, aggregation is far less simple and require a user defined method. User defined methods for both types of data are provided by Quince actors, described in the analysis section. The motivation for providing a method to aggregate string data is in order to preserve relevant logs, events and other textual data, which may have future use, beyond the aggregation interval of the receiving buffer. Full, unaggregated log storage is beyond the purview of BufferTable as it prioritises recent data over older data. Numerous log storage systems including Splunk, Logstash and syslog-ng exist which can store logs indefinitely. Much how BufferTable can push state to a database or flat file, so too can it be configured to commit state to a log storage facility.

Chronologically ordered metrics can be inserted into the ring buffer in constant time. An out of order metric is entered into the ring buffer using a binary search to locate the index which falls within the appropriate time period, separates the buffer into two slices at that index, appends the metric to the lower slice and then appends the upper slice. This results in a $O(logn)$ performance for out of order inserts.

A BufferTable instance is provided with an upper memory bound which it will not exceed. When instantiated the BufferTable is empty. Buffer families are created when metrics are submitted to the the BufferTable. Initially, buffer families are created with an equal share of the BufferTable's allocated memory. Within a buffer family, memory is allocated to buffers in a logarithmically decreasing manner from the receiving buffer to the final aggregate buffer. If buffers with larger intervals are empty, their capacity is subdivided amongst the preceding buffers. Should a buffer family reach its assigned capacity it will employ a LIFO strategy whereby aggregated metrics from the final buffer will be discarded. To avoid the loss of these metrics, a persistent storage mechanism must be used as timely collection and analysis of monitoring is the primary use case of Varanus and not long term storage.

Varanus analysis service

The large set of potential monitoring use cases necessitates analysis tools which provide a vast range of functionality. MapReduce has long been cited as the killer application of cloud computing and has become the de facto means of expressing data intensive computation. MapReduce was however intended for batch computation and as such gives no time guarantees nor does it guarantee that data will be used as soon as it is available. This has led to a

range of MapReduce and MapReduce-like services which attempt to support real time computation. The most popular of these is Apache Storm. Storm, MapReduce and indeed most other tools of the domain are intended to run over arbitrary clusters and are not indeed to run on shared use hosts. This makes the use of MapReduce and its contemporaries inappropriate in Varanus as it is built around the goals of resource awareness and being secondary to other applications running on VMs. Rather than attempting to develop yet another MapReduce implementation which meets the goals of Varanus it is preferable to instead provide a means to export Varanus state to an existing MapReduce implementation, if MapReduce is indeed required.

In lieu of MapReduce, Varanus exposes an alternative processing framework which is better suited to analysing monitoring data. The primary means of analysing monitoring state in Varanus is Quince [4] an event based programming model which in the same vein as other Varanus services, distributed computation over the entire cloud deployment. Quince is intended to allow developers to build upon Varanus and to develop new features and implement autonomic failure detection and correction strategies. The actor model was chosen as it maps well to the underlying Varanus architecture and allows actors to be relocated and message routing to be altered according to resource usage, membership change and other phenomena with ease, where as other concurrency models would not support this as easily.

Quince

Quince is an event based programming model to support real time monitoring data analysis and the development of autonomic workflows. Quince expresses computation in terms of events and actors which process those events. The actor model is a well researched model of concurrent computation which is now seeing application in the cloud domain. Actors provide a high level, location independent model of concurrent programming which scale out and scale up with relative ease. Actors, therefore make an ideal basis for facilitating various forms of autonomic programming.

Quince exposes three units of computation to end users: sensors, events and actors. The user expresses their computation in the form of sensors, which generate events and actors, which consume events. This orchestration is intended to facilitate autonomic programming with the sensor and event-actor chain intended to directly map to the monitor-analyse-plan-execute autonomic pipeline. Additionally, while more batch computational models such as MapReduce support only acyclical workflows, Quince has no such limitation making it a superior choice for real time processing.

Sensors are written by the user and run not as part of the analysis service but rather as part of the storage service. Sensors are associated with values in the storage service and run periodically when values are updated. Should values in the state store meet the conditions specified in a sensor be met an event is generated. Sensors provide a means to regularly check conditions and trigger an event, for use cases where regular checking is not needed events can be generated without a sensor. This allows events to be generated natively within applications, rather than having a sensor check state collected from the application. Not all use cases however require a sensor, nor a complex event. The sensor is, therefore, entirely optional. Actors can be called though a simple API call via any software. Additionally events can empty aside from the type field, allowing events to serve as simple messages to call actors.

Once events have been generated they are then distributed. Event distributed occurs in one or more of the following manners:

1. Local handling, the event is passed to an appropriate actor (if one exists) at the local Quince runtime.
2. Direct delivery, the event is sent to one or more specifically designated hosts. Hosts are designated by the coordination service. By default the least loaded hosts are prioritised for direct distribution.
3. Propagation, the event is propagated to to the entire deployment or logical subgroup via a hierarchical gossip broadcast.

These three distribution mechanisms allow events to be processed locally for efficiency, sent to a dedicated event processing instance for consistency or propagated over the deployment to perform complex, distributed tasks and analysis.

Actors are also written by the end user and are called when a Quince runtime receives an event appropriate to the actor. If an event is received and no appropriate actor is present the Quince runtime will either disregard or forward the message according to what is specified in the event. When invoked the matching actor will decode the event, perform their prescribed action and optionally generate an additional event. This allows for the creation of event chains, whereby a string of actors are invoked though the use of intermediate events.

In addition to regular actors Quince provides the notion of preprocessors. Preprocessors are invoked by being specified in the preprocessor event field and are called before the regular actor. The result of the preprocessor, if any, is what is passed to the regular actor. Preprocessors are otherwise identical to regular actors, differing only in their method of invocation. Quince by default, provides three preprocessors:

1. Aggregate. The aggregate actor takes two parameters, a delay and a aggregation method (count, max, min, medium, mode or sum). All events received from the initial invocation until he delay has elapsed are aggregated according to the specified method and a single event is produced as a result.
2. Filter. The filter actor takes two parameters, an event field and a search pattern. If the search pattern does not match the given field the event it is discarded. In addition to regular expressions the pattern can search for the special values *last* and *expected* which correlate to state store values representing the previously recorded state and the normal state.
3. Redirect. The redirect actor takes an argument specifying which distribution method (listed above) to redistributed the message on.

By combining actors and preprocessors developers can create powerful event chains which can detect notable states and phenomena and in turn implement corrective action. The primary motivation behind this problem is to provide a scalable, distributed means to execute monitoring workflows. Using this programming model it is near trivial to implement basic monitoring practices such as threshold checking and feasible to implement complex autonomic monitoring workflows.

Actors in Quince are not bound to a physical host, Quince actors exist in *virtual actor space* an abstraction which maps a sub-region. Individual actors are instantiated within the actor space and are identified by 3 values: a type signature, a instance ID and locator, signifying the VM currently hosting the actor. In the virtual actor space, at any given time, there can be zero, one or many instance of an actor. Serialised actors are stored as immutable objects within the coordination service configuration store and are instantiated when necessary. Actors are dynamically loaded when an event handled by a non-instantiated actor is raised. Actors are instantiated to hosts according to the least loaded strategy used throughout Varanus. The hosts which act as Quince runtimes and which have instances of each actor in the system are announced via the configuration store.

Computation placement

While most current actor implementations provide location agnosticism, whereby actors are invoked locally or remotely using the same semantics, most do not decouple the actor from an endpoint. Current actor schemes predominantly rely upon hardcoded URIs or some external lookup service such as DNS or JNDI to provide a URI. This scheme ties actor instances to a specific host and requires all clients to update their URIs in the case of failure or configuration alternation or rely upon some form of indirection. This is non ideal and runs contrary to the notion of rapid elasticity and simple horizontal scaling. As the hosts which comprise a cloud deployment are prone to rapid (and possibly frequent) change it is not preferable to have actors tied to a specific host. Rather it is preferable to have actors able to move between hosts and sets of hosts transparently and encapsulate this into the existing notion of location agnosticism.

Unlike previous actor implementations actors in Quince are not bound to a physical host. Quince actors exist in a *stage* otherwise known as a *virtual actor space* an abstraction which maps to the underlying resources which comprise the cloud deployment. Individual actors are instantiated within the actor space and are identified by 3 values: a type signature, a instance ID and locator, signifying the VM currently hosting the actor. In the virtual actor space, at any given time, there can be zero, one or many instance of an actor. The virtual actor space maps to the underlying resources. Specifically it maps to two locations: an actor store and the Quince runtime.

The actor store is any data store which supports the hosting of packaged code archives (which is virtually all of them). In our implementation the actor store is provided by a Kelips [18] like DHT which is also hosted by the VMs within the deployment. The actor store could, however, be an external database or service such as Amazon S3 or equivalent. Non instantiated or *cold* actors reside in the actor store until required. They are dynamically loaded when a request to a cold actor is issued. In our implementation this is achieved through Scala/JVM dynamic class loading which fetches the respective jar file containing the packaged actor front the actor store to an appropriate VM, instantiates the actor and then handles the incoming request. Instantiated or *hot* actors are hosted by Quince runtimes which reside upon each VM within the deployment. This scheme could also allows for hot actors, including their state to be serialised to the actor store.

Actors are instantiated on specific VMs within the deployment based upon demand. Due to the manner in which actors express computation and the message passing therein, parallelising and concurrently executing actors is incredibly simple. For this reason, it is trivial to have multiple independent actor institutions on the same or on multiple hosts. The mechanism which decides where and how many actors to instantiate is the actor coordinator. The coordinator runs as part of each runtime and is itself an actor, as such it can be easily substituted for alternative implementations. The default coordinator decide where and how many actors to instantiate based upon four factors: the number of events per second currently being generated, the historical volume of events, the response time of current actors and resource usage of VMs within the deployment. All of this data is made available via the Varanus monitoring tool. The coordinator uses this state in conjunction with a series of simple

linear regressions to estimate the appropriate number of actors.

As actors can be moved or terminated there is need to maintain knowledge of the location of actors. Quince tracks the location of actors by distributing a lookup map to each runtime which specifies the current location of each instantiated actor. Each Quince runtime maintains a map (in our implementation this is a Guava multi map) which maps actors to endpoints which host actor instances. New actor instantiations register themselves with the local runtimes's map which is then disseminated to all other runtimes within the deployment. Similarly actor instances on VMs which are undergoing termination mark themselves in a tombstone state prior to termination and then eventually expunged from each runtime's lookup map. If the VM fails or is unable to register the termination it then becomes the prerogative of the failure detector to remove the dead actor from the lookup map.

The lookup map is disseminated and kept consistent between all runtimes either via a pub-sub for client runtimes or gossip scheme for actor hosting runtimes. Both these methods of synchronisation achieve eventual consistency, therefore there is the possibility that calls may be made to no longer existing endpoints or runtimes may not yet be aware of newly instantiated endpoints. Again, the circuit breaker mechanism is intended to lessen the significant of this issue.

The mechanism described above affords Quince the mean to instantiate, terminate and move actors on an ad hoc basis. What is required to use this mechanism is a policy that stipulates where actors are placed and when to change their placement. In the current Quince implementation this is achieved through a simple heuristic approach. Quince deployments are instantiated with a heuristic map which associates a given condition with an action. This map, by default, contains heuristics to instantiate additional VMs when the actor system has insufficient resource, to terminate VMs when there are excessive resources for a significant period and a series of heuristics to increase and reduce actor instantiations according to demand. This basic approach is not guaranteed to achieve an optimum solution to the problem of computation placement and can result in sub optimal decision making. A constraints or machine learning approach may be superior and can be slotted in as a substitute with relative ease.

Example 1

Consider the problem of concatenating related log entries from a distributed application that are generated within 10 min of each other and then enumerating how many concatenated logs were produced within the 10 min interval. Each log is identified by an ID and log entries are not guaranteed to be causally ordered. Figure 9 provides a Scala implementation of this scheme.

In this example there is a sensor, `LogSensor` and two actors: `join` and `report`. LogSensor periodically checks for new log entries, filter the relevant log type and passes it to join. `join` extracts the ID from the new log and makes an API call to a document store to ascertain if another log entry with the same ID exits. If it does, it concatenates the two entries and pushes the result to the document store. If not, it commits the partial entry to the document store and awaits the next entry. `report` is called in conjunction with the aggregate preprocessor to enumerate and report to a service the number of log entries concatenated within the 10 min window.

Example 2

Consider the problem of detecting high CPU usage then if the rest of the *sub-region* is heavily loaded, start additional VMs to compensate for load. The user would write a sensor and actor similar to that in Fig. 10.

In this simple example the sensor `CPUSensor` is executed by the monitoring agent when the local CPU usage is updated in the state store. Should the CPU capacity exceed 90 % the sensor generates a `cpuEvent`, and specifies direct distribution. This will propagate events to a specifically designated host, by default the coordination

```
class LogSensor(val log : String) extends Sensor:
        def run() = {
                        preprocessor = new Filter(
                             value, "myApplication")
                        trigger(preprocessor(new
                             JoinEvent(log, direct))
                }}

class Join(val log : LogEvent) extends Actor{
        def run() = {
                val IDRegex = new Regex("ID:\d{5}")
                val ID = (pattern findAllIn str).
                     mkString())

                val existingLog = documentStore.get(
                     ID)
                if (existingLog){
                        val concatLog = existingLog
                             + logEvent.value
                        documentStore.put(ID,
                             contactLog)
                        preprocessor = new
                             Aggregator(600, count)
                        trigger(preprocessor(new
                             ReportEvent(1, direct))
                } else {
                        documentStore.put(ID,
                             logEvent.value)

        }}


class Report(val notification : JoinEvent) extends
     Actor{
        def run() = {
                reportingService.report(notification
                     .value)
        }}
```

Fig. 9 Concatenating logs using Quince Actors

```
class CPUSensor(val cpu : Int) extends Sensor:
        def run() = {
                if (cpu > 0.9){
                        preprocessor = new
                            Aggregator(60, count)
                        trigger(preprocessor(new
                            CPUEvent(cpu, direct))
                }}}

class CPUactor(val cpuEvent : CPUEvent) extends
     Actor{
        def run() = {
                val = numHosts = varanus.store.get(
                     hosts).size()
                if(cpuEvent.value >= (numHosts / 2))
                     {
                        startNewVM()
                }}}
```

Fig. 10 Detecting and mitigating high CPU usage using Quince Actors

services designates the least loaded host. Prior to invoking the actor, the event is first passed to a preprocessor which aggregates the values of all events of the cpuEvent type received over a 60 s period. The aggregate event is then sent to the `CPUactor` which uses the value contained in the event (which after the aggregate sub-actor is the number of loaded hosts). The actor, `CPUactor` consumes the aggregate event and establishes what proportion of the VM deployment is loaded. If more than half of the VMs within the deployment are at 90 % or greater CPU utilisation, a new VM is started.

Evaluation

Evaluation architecture

Deploying distributed applications is often challenging due to the logical and geographic separation of components. Scale often exacerbates this problem, producing a volume of work which far surpasses what humans can do in a timely fashion. Common cloud computing best practice stipulates that SSH (or RDP in the case of Windows VMs) is a last resort. That is that a human should only log in to a VM in the event that of some unanticipated behaviour which cannot be handled through automated tools.

The evaluation of Varanus was performed using Amazon Web Service, therefore instead of using generic cloud computing terms this paper will use AWS specific nomenclature. EC2, like most IaaS clouds, allow users to seed an executable script to a VM on instantiation. This is the mechanism which is used to provide the basic configuration to each VM, necessary to obtain all the relevant software and to deploy the configuration. This script is transmitted to VMs and once they have finished the boot process, the script is then executed.

The VMs in our evaluation all run Ubuntu 12.04, instantiated from the Amazon Machine Image ami-ecda6c84. Each VM instance is seeded a very simple Bash script which installs Puppet and then uses Puppet to install and configure all subsequent software. Puppet[5] is an Open Source configuration management tool which provides a declarative Ruby based DSL which enables users to build *manifests* which describes software installation, commands and configuration which should be performed on clients. Puppet is used to install Varanus and the necessary software to collect data and run experiments and give Varanus instances the necessary bootstrap information.

In addition to the deployment which is being evaluated, additional VMs are provisioned for services which support the experiments. These are:

The Results Service: a simple message service endpoint which provides a mechanism for clients to push their experimental results. This service is written in Python and uses a ZeroMQ pull socket to act as a sink which waits for results to be pushed. In addition to Varanus, Puppet installs a client which pushes results to the result service once the experiment has completed.
The Events Service: the events service is a mechanism to simulate faults and to terminate and instantiate VMs. The events service takes configuration files written in JSON which specify what events to trigger and when. Events include: terminate an arbitrary number of VMs, terminate a specific VM, instantiate a number of VMs, trigger a failure and instruct Puppet to pull down configuration to VMs. In order to achieve this, the events service communicates with the EC2 API in order to instantiate and terminate VMs and with a small client on VMs in order to simulate failure and run arbitrary commands. The events service is written in Python and uses ZeroMQ push sockets for messaging.
The Puppet Master: the Puppet master serves and compiles manifest files which specify the configuration for VMs.

This architecture is described in Fig. 11. When evaluating larger numbers of VMs the results service and the Puppet master become overloaded and hamper the experiment. Thus when using larger number of VMs they are replicated over multiple VMs which are round-robined between using the Route 53 DNS service in order for them to scale and meet demand.

Comparison against current monitoring tools

It is difficult to compare like for like Varanus against other real world tools. Most real world monitoring tools do not expose mechanisms to examine their own performance and behaviour and as such would require significant modification. Additionally, many of these tools do not have out of the box support for cloud environments and would necessitate the devising of a means to deploy and manage them on the cloud. Despite the difficulty in doing so, we deemed it essential to examine Varanus against at least

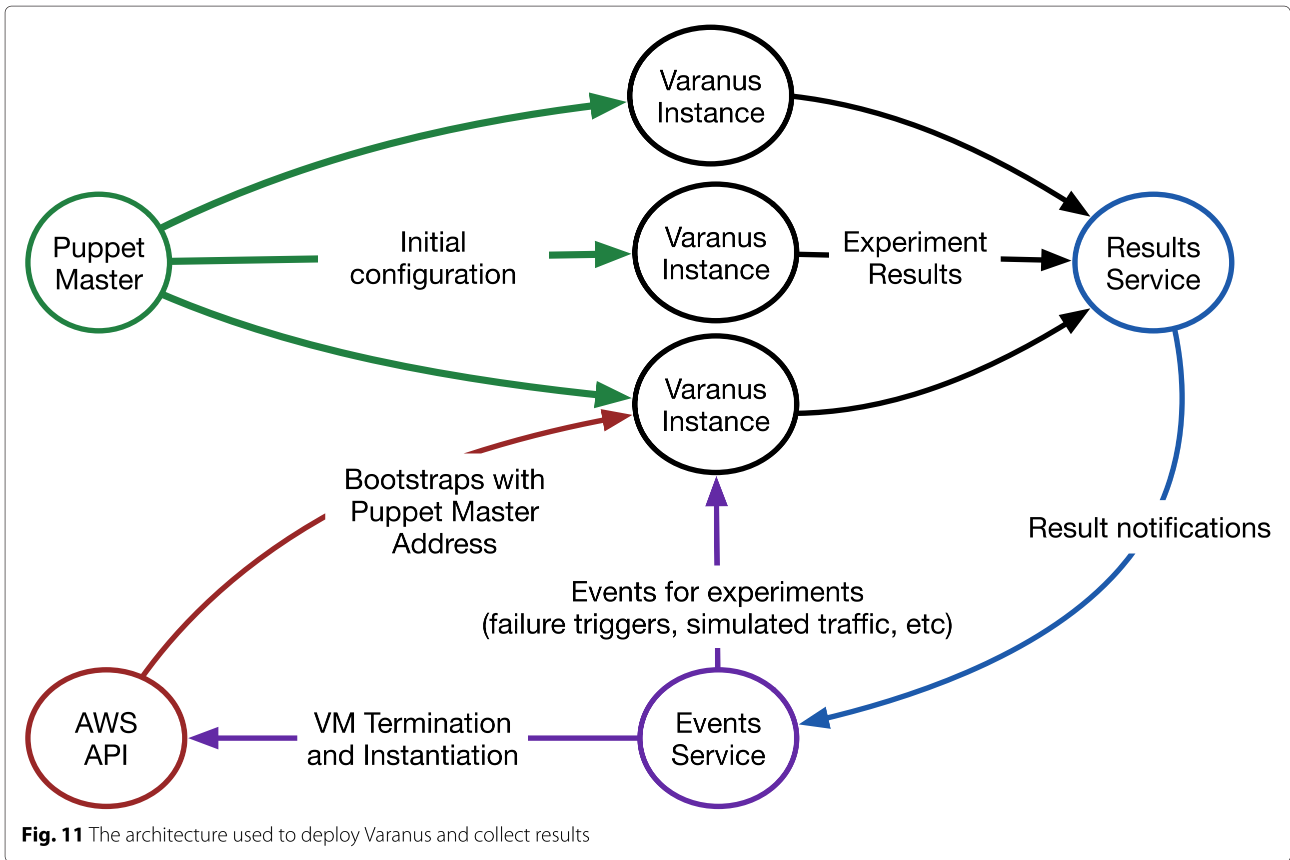

Fig. 11 The architecture used to deploy Varanus and collect results

once real world monitoring tool. Therefore, we evaluate Varanus against Nagios - the world's most popular Open Source monitoring tool. To perform a more thorough evaluation than a comparison against a single tool, we have devised a novel compromise. In addition to Nagios, we compare Varanus against our own implementations of monitoring architectures which are found in common monitoring tools. These implementations use the same libraries and share some code with Varanus and therefore allow for an unbiased comparison of the underlying architecture rather than being influenced by implementation details.

Common monitoring architectures

There are a common set of architectures which underpin many current tools. In our evaluation of Varanus we consider our own implementations of these architectures in lieu of further examining existing tools. In our attempts to examine current tools like for like, we found that differences in encoding formats, networking and other secondary features prevented an unbiased examination. This is evident from our comparison against Nagios whereby there are disproportional disparities between the performance of Varanus and Nagios. This disparity is certainly in part due to the inappropriateness of Varanus to cloud monitoring but is exacerbated by the discrepancies in implementations. Furthermore, many popular monitoring tools have large codebases which are poorly documented and do not easily lend themselves to experimentation. We therefore implemented a number of common monitoring architectures using the same networking libraries, encoding format, language and other factors as per Varanus to provide an effective like for like comparison. The intention behind this is to see how the architecture of Varanus fairs against existing architectures which are implemented using modern libraries and benefit from awareness of cloud properties.

These architectures include:

- 2-tier push model. This is the architecture employed by collectd, statsD and many monitoring as a service tools. Monitored clients are provided the address of a monitoring server to which they push monitoring state at their own prerogative. The single central monitoring server acts as a sink; receiving monitoring state and analysing the incoming data.
- n-tier Push Model. This is similar to the architecture employed by Collectd and an extension of the 2-tier push model. Rather than pushing directly to the top level server monitored VMs push state to an

intermediate monitoring server. The intermediary, performs analysis on the monitoring state and in turn transmits the outcome of the analysis to the top level monitoring server. This forms a three level tree. In real world implementations of this architecture the tree can have arbitrary depth, with several levels of intermediary monitoring servers. This is not, however, common and thus our implementation is of the typical 3-tier variety. This design is a more scalable variant of the 2-tier push model and alleviates much of the pressure on the top level monitoring server. The top level server is however still responsible for assigning monitoring servers to clients, aggregating state from monitoring servers and still receives significant volumes of messages. We evaluate two variants of this scheme: a fixed and a variable version. The fixed version uses 11 VMs, 1 as the root monitoring server and 10 as intermediaries. This variant is still quite common in real world tools, despite the fact that it has no way to accommodate for elasticity. The second variant, instantiates additional VMs as is necessary; doing so when the current pool of monitoring servers is overloaded.

- 2-tier pull model. This is the architecture employed by basic 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. When new VMs are instantiated, this schedule must be recomputed in order to include the new VM. All collection and analysis of monitoring data is done on a single VM. This is the oldest of the commonly employed monitoring architectures and tightly couples components. It is entirely expected that this will be amongst the least scalable architectures.
- 2-tier pull model. This architecture can be used by Nagios, Xymon, Cacti, Ganglia and is an extension of the 2-tier model. This architecture is similar in concept to the 2-tier push model however clients are not in control transmitting monitoring state to servers. Instead, monitoring servers generate a polling schedule and poll clients according to that schedule. Intermediate monitoring servers collect and analyse state. The top level server then polls the intermediate servers, aggregating state and the results of the analysis. This, much like the 2-tier model, requires the recomputation of the schedule when sub-tree membership changes. Once again, there are two variants of this model: the elasticity intolerant fixed version and the variable version which can provision additional VMs as necessary.

Each of these monitoring architectures is implemented using the same libraries as Varanus: using ZeroMQ to transmit data and using protocol buffers to encode state on the wire. The Varanus collection service (with a little modification) is reused as the monitoring client for each of these architectures. It collects the full gamut of system statistics and a full set of Apache metrics. In total this results in 200 metrics. As for analysis, only simple threshold analysis is done. Each of the monitoring architectures (and Varanus) are provided a set of thresholds to correspond with the metrics and checks for threshold violations each time monitoring state is received.

Elasticity

Central to our claims regarding the effectiveness of Varanus as a cloud monitoring tool is the notion of elasticity. Elasticity: the propensity for cloud VMs to rapidly change in scale and composition is problematic for many monitoring activities. Elasticity has significant implications for a monitoring system and can disrupt services, interfere with failure detection, introduce latencies and incur computational costs. Any monitoring tool well suited for cloud monitoring must therefore tolerate elasticity.

Conceptually, elasticity is similar to the notion of churn [21] from the domain of peer-to-peer computing. Given closer examination, however, the two concepts differ. Notably churn encompasses near constant change in membership, short-lived membership and the reappearance of previously seen peers. Elasticity differs in that: membership change typically occurs infrequently but involves a significant portion of all members, members will typically have at least an hour uptime (due to the common practice of billing in hour increments), and members are not capable of rejoining (due to VM termination). It is therefore infeasible to rely upon previous categorisations of churn in order to understand elasticity.

Elasticity describes the instantiation and termination of resources to meet demand. This encompasses compute, storage, network and other resources. With regards to monitoring, we are predominantly concerned with VMs. By far the most common mechanism used to control elasticity and alter the scale and composition of cloud deployments is auto scaling. Each cloud provider has their own implementation of the concept, each with different semantics but the underlying concept is the same. An auto scaler instantiates or terminates VMs as application demand changes.

One of the most common use-cases for cloud computing by far is he hosting of web applications. A web application utilising an auto scaling tool will change in scale based upon http requests. An auto scaler will either consider resource usage on the existing web servers (which increases proportional to requests) or the volume of requests alone. Either way, by examining http request patterns we are able to understand to some degree

the patterns of elasticity encountered by common hosted cloud applications.

Pursuant to this goal Fig. 12 shows the hourly volume of http requests over a 24 h period to 5 low to high traffic web servers. This data is obtained from publicly available server traces namely those from NASA [22], the EPA [23], The University of Saskatchewan [24] and Clarknet [25]. The graphs shown in Fig. 12 show a subset of those traces and demonstrate some of the common trends.

Three of the common patterns (that are relevant to elasticity) which emerge from the http traces are exponential and linear changes in traffic and constant traffic:

- An exponential increase in traffic is common when access patterns are related to the time of day. Many business orientated applications see an exponential increase in demand during opening of business and a logarithmic decay once trading hours have ended.

Fig. 12 Examples of traces of HTTP requests per hour over a 24 h period. Taken from a number of common public sources. Shown here are traces from the University of Saskatchewan, NASA and ClarkNet

This pattern is also typical of applications and resources which are made popular via social media. As the resource becomes popular demand exponentially increases and as it fades from vogue there is a logarithmic decrease. Exponential change in load necessitates a rapid change in the cloud deployment in order to either code with demand or to avoid overpaying for under-utilised resources.

- A linear change in traffic is more common in web applications which are not as associated with time and while being notable does not require the same dramatic change in deployment composition but will still require some alteration.
- Exactly constant traffic is quite unlikely, however for the purposes of elasticity periods where there is not a sufficiently large change in load is for all intents and purposes considered constant. This is where the provisioned resources are sufficient to meet demand and do not go beyond what is required.

The exact change in deployment composition is based upon an administrator defined rule. Common rules include adding an additional VM each time an existing VM exceeds 80 % cpu utilisation, instantiating additional VMs per a given number of requests, and instantiating additional VMs while application latency remains above a given threshold. The rules for terminating VMs are then a sensible reversal of the previous rules.

Based upon a trivial benchmarking exercise using the Apache stress testing tool of EC2 m1.medium VMs it is simple to examine the first two rules. Based upon the stress testing of 10 m1.medium instances the mean maximum requests per second an instance could fulfil was 2431 requests per second. 80 % CPU utilisation was exceeded upon handling around 2000 requests per second. Given a satisfactory leeway one can generalise that after a sustained period of greater than 2000 requests per second an additional VM should be instantiated. Many auto scalers specific the sustained period as a 15 min window.

By experimenting with this generalised rule in conjunction with the Amazon EC2 Auto Scaler and recreating portions of the http traces using Apache JMeter we were able to derive models simulating linear and exponential periods of elasticity which resemble what real world web applications encounter.

Monitoring latency

Monitoring latency is the time between the collection of subsequent monitoring state. Monitoring latency is separate from but related to the collection interval (which defines the frequency at which effort is made to obtain data). In the vast majority of monitoring tools the collection interval entails a best effort scheduling, in an ideal environment the monitoring latency should equal the collection interval, however this is seldom the case. Network latency, loaded hosts or other phenomena can serve to increase monitoring latency. This is inherently undesirable as it creates 'blind spots' whereby administrators or autonomic agents are unaware of the state of the system and thus unable to take any requisite actions to modify the system.

Historical best practice recommended between 1 and 5 minutely intervals for the collection of most variables. More recent best practice such as that advised by CopperEgg [26] and DataDog [27] reduces the collection interval to one second for all frequently changing variables. A modern cloud monitoring tool must therefore be capable of propagating monitoring state at an interval as close to one second as is possible.

Linearly increasing deployment size

This initial set of experiments act as a precursor to our investigation into the effects of elasticity upon Varanus and other monitoring tools. In this scenario, Varanus and other tools are provisioned in deployments of increasing size. Once these deployments are provisioned and are stable, monitoring latency is recorded. There is no bonafide elasticity, once deployments are provisioned they do not change until termination. This allows the effects of scale to be investigated prior to the effects of elasticity.

Experiment 1

In this experiment, we test Varanus and Nagios using the previously described methodology. Figure 13 shows the monitoring latency of Varanus and Nagios over deployments ranging from 100 VMs to 5000 VMs. As is clear, Nagios (even the modified version) suffers from linearly increasing monitoring latency, proportional to the size

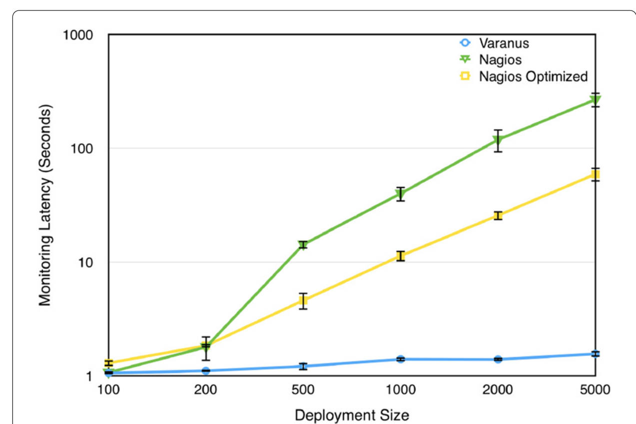

Fig. 13 Monitoring latency with varying deployment size. Here it is evident that Varanus maintains a low, near 1 s monitoring latency where as both versions of Nagios incur near linear increases in latency

of the monitored deployment. This is due to the manner in which Nagios performs data collection: it generates a schedule and linearly iterates through that schedule until all polling has been completed for that time period. As the size of the schedule increases as does the time required to iterate through them. In the optimised version of Nagios, additional concurrent threads are used to perform polling and a number of patches have been applied which decrease the time of each poll. Despite this, as the size of the deployment increases both versions of Nagios suffer from increased monitoring latency with the stock install reaching a mean latency of 266 s and the modified version reaching mean of 58 s. The second value is consistent with the historic best practice of minutely data collection but is far from the per second interval that is desired.

Varanus, meanwhile is an entirely push based architecture with no centralised scheduling. As such Varanus maintains a monitoring latency much closer to the desired one second. In the case of a 5000 VM deployment Varanus achieves a mean latency of 1.64 s with an upper bound of 2.04 s. The upper bound is due to the a Varanus storage service instance suffering from higher than average load. This is rectified by distributing clients between storage instances. If there is an insufficient number of underloaded servers in the deployment, Various monitoring latency will increase. Figure 14 compares Varanus running over a semi loaded deployment with Varanus running over a semi loaded deployment with additional dedicated, unloaded servers. In this case Varanus maintains a mean monitoring latency of 1.07 s when provided with additional dedicated servers.

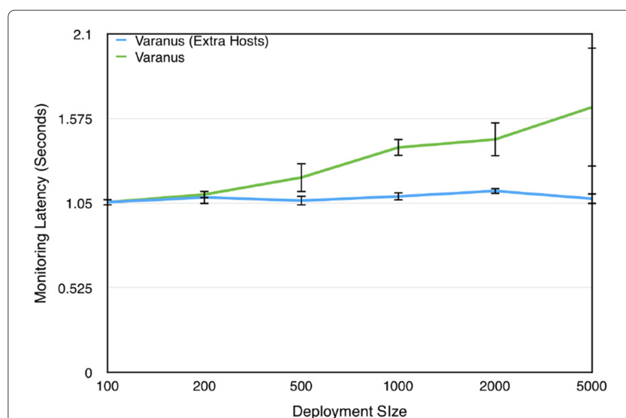

Fig. 14 Varanus monitoring latency with and without additional Dedicated Hosts in a loaded deployment. As Varanus attempts to use the existing resources in a a cloud deployment its performance depends upon the existence of under-utilised resources. If resources are at a premium, Varanus performance will suffer. This is particularly evident when at scale as typified by the difference in value and std deviation at a scale of 5000 VMs between Varanus with and without additional dedicated hosts

Experiment 2

In this experiment, we repeat the previous experiment this time comparing Varanus against other monitoring architectures. Figure 15(a) shows the monitoring latency of Varanus and the push/pull variations of 2-tier and n-tier monitoring architectures. As is clear, Varanus once again maintains a low, near constant monitoring latency where as other monitoring architectures suffer increased monitoring latency as scale increases. Of particular note is the behaviour of several monitoring architectures at the 1000 VM point. Figure 15(a) show the same results as Fig. 15(a) presented using a logarithmic scale. From this it is clear that many existing tools maintain appropriately low monitoring latencies at the 500 VM point, however at 1000 VMs there is a sudden increase in monitoring latency. This is due to the VM at the top of the hierarchy suffering heavy load and thus processing monitoring data at a slowed rate. An example of this is the central VM in the n-tier push fixed size architecture which reported a 5 min load average of 138 at a scale of 500 VMs. The logarithmic

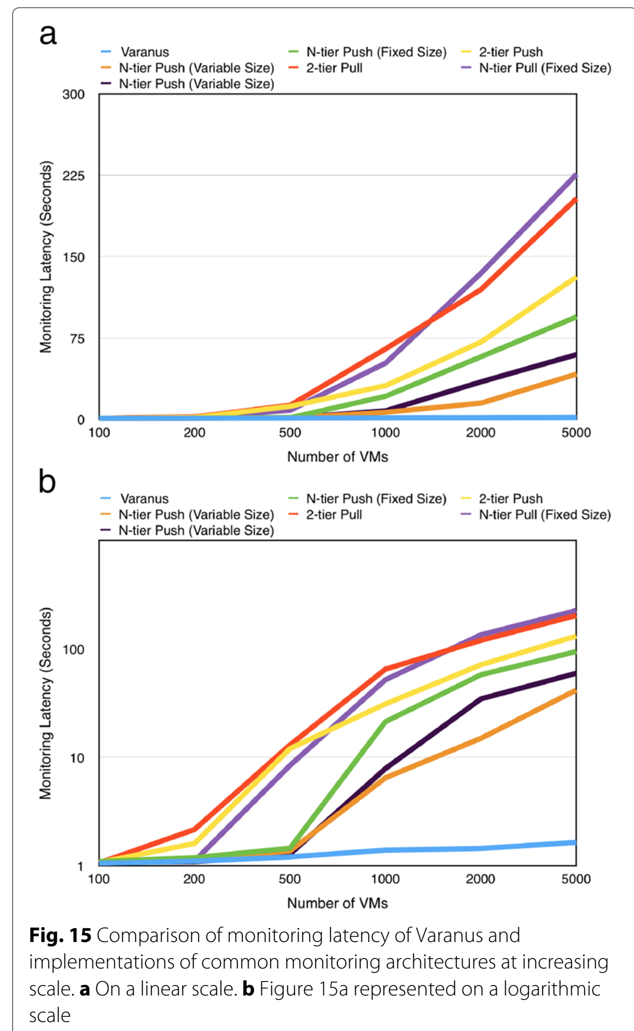

Fig. 15 Comparison of monitoring latency of Varanus and implementations of common monitoring architectures at increasing scale. **a** On a linear scale. **b** Figure 15a represented on a logarithmic scale

scale also highlights how Varanus deviates from the goal of 1 s monitoring latency as the scale of the deployment increases. At 5000 VMs, Varanus incurs an average monitoring latency of 1.65 s. Again, this could be reduced by provisioning additional dedicated VMs.

Deployment size varying with elasticity

The previous experiment elucidated upon the effects of continually increasing deployment size on monitoring latency. What was not taken into account, however, was VM termination. In this set of experiments, the deployment is subjected to varying levels of load. This is more natural and is much more typical of what is found in real world scenarios. Here, VMs are provisioned as load increase and terminated when load subsides. This is rather more arduous for monitoring tools compared to the previous scenario.

Experiment 1

Figures 16b and 16a show the effect of elasticity upon monitoring latency. In both these cases, web server trace data is used to produce realistic fluctuations in demand which is in turn used to induce elasticity via the use of the AWS Elastic Load Balancer. Figure 16b shows the results of a moderate, sustained traffic (starting at 10,000 requests and gradually increasing to 1,000,000 requests per second and then gradually subsiding to 10,000 requests per second) representing what a popular web site may receive on an average day. Figure 16a shows the results of a sudden spike in traffic (starting at 10,000 requests per second and increasing to 10,000,000 requests per second) representing what a web site may encounter in a special case (e.g. sudden popularity on social media, a special event, etc). This latterly case is an edge case, however it is one of the key use cases of cloud computing. It is therefore essential that cloud monitoring tools be able to tolerate sudden peaks in load without suffering from QoS degradation.

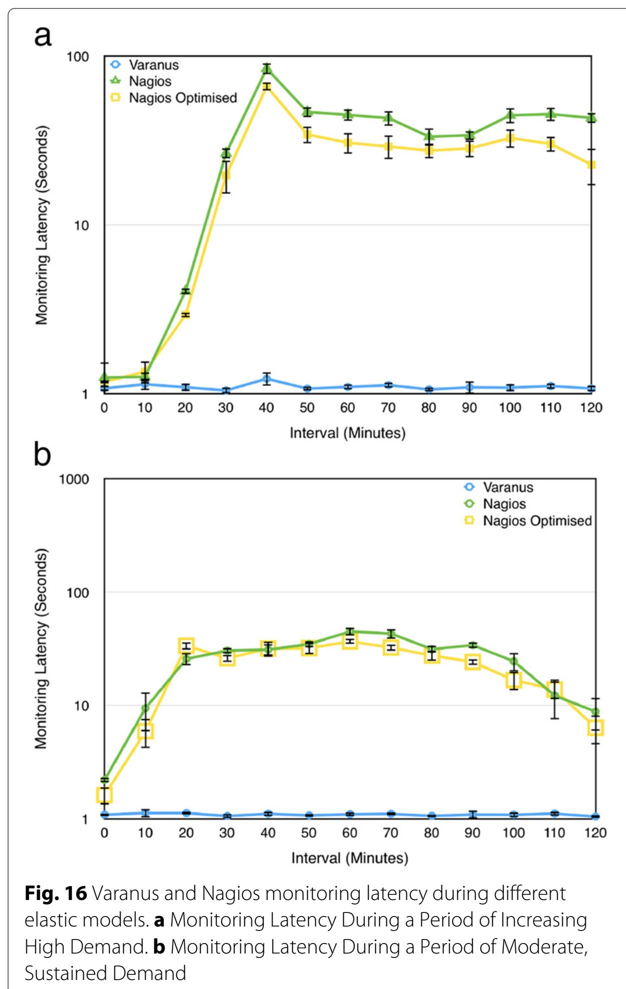

Fig. 16 Varanus and Nagios monitoring latency during different elastic models. **a** Monitoring Latency During a Period of Increasing High Demand. **b** Monitoring Latency During a Period of Moderate, Sustained Demand

Nagios demonstrates a significant increase in monitoring latency as additional hosts are instantiated to meet demand. In Fig. 16a Nagios provides low latency monitoring until the sudden increase in demand causes the instantiation of additional VMs. As VMs are continuously instantiated until demand is met Nagios exhibits a sudden spike in monitoring latency. Part of this spike is attributable in the bootstrapping of new VMs, which requires Nagios to restart and thus further increases monitoring latency. Which levels off after the number of VM becomes constant.

Again, as Varanus makes use of a push mechanism and has a scalable bootstrapping mechanism elasticity is tolerate and has limited effect upon monitoring latency. Much like the result in the previous experiment, Varanus achieves a low monitoring latency which in its worst instance reaches 3.1 s.

Experiment 2

Again, we repeat experiment 1 this time with our own implementations of common monitoring architectures. Figures 17a and 17b show the effects of two elasticity model on monitoring latency. Producing results not dissimilar to the real world Nagios, none of the monitoring architectures maintain a consistently low monitoring latency while encountering either elasticity model. In the case of the moderate load, the majority of the monitoring architectures encounters a near linear increase in monitoring latency as requests increase. The two variable architectures perform better than the others but each still suffer from increased monitoring latency. Despite being able to provision additional VMs in order to tolerate demand, the tree based structure of each of the variable architectures resulted in top level VM eventually subsuming to load and monitoring latency increased as a result. This is also the case in the high demand elastic model, where monitoring latency in the monitoring architecture implementations suffers significantly due to the

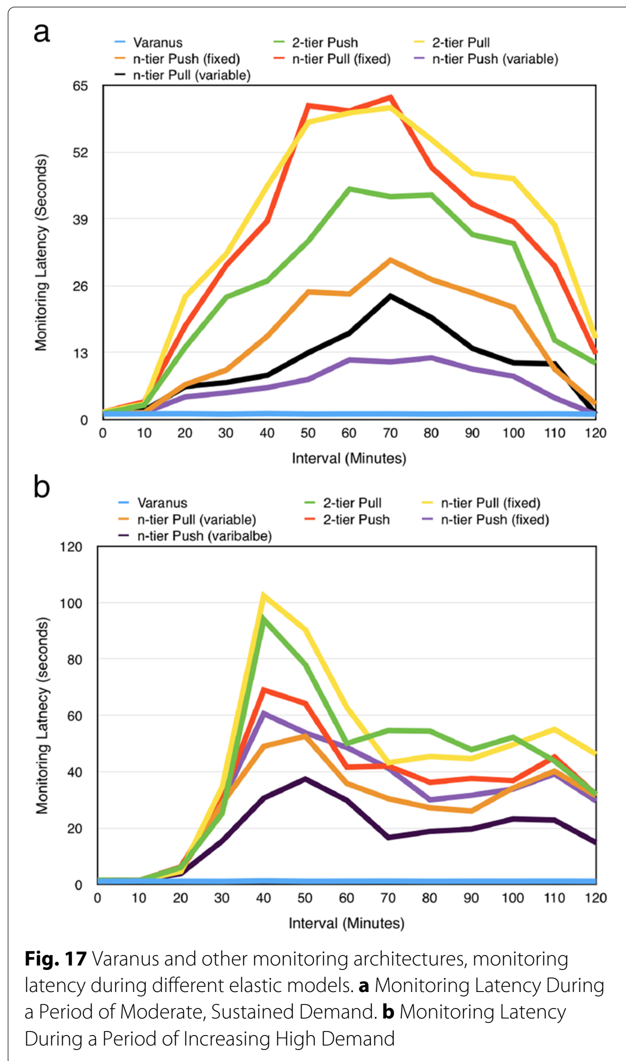

Fig. 17 Varanus and other monitoring architectures, monitoring latency during different elastic models. **a** Monitoring Latency During a Period of Moderate, Sustained Demand. **b** Monitoring Latency During a Period of Increasing High Demand

high levels of load encountered. In each of the monitoring architectures, the top level VMs suffered high CPU, network and memory usage as a result of the large volume of messages being send from monitored VMs or VMs further down the hierarchy. Varanus, owing to its decentralised architecture does not heavily load individual VMs, rather it amortises load over the deployment - allowing monitoring latency to be kept low.

Investigating resource usage

A priori, lower resource usage is desirable. A monitoring tool which requires significant resources will have high fiscal costs and risks interfering with the deployment that it is monitoring. The primary resources consumed by monitoring tools is CPU, while monitoring can be data and thus network intensive, it is the aggregation, analysis and handling of monitoring data which is typically the bottleneck - all of which is CPU bound work. The following set of experiments investigate the CPU usage of Varanus, Nagios and other monitoring architectures and how this impacts their appropriateness for cloud monitoring. Specifically, we investigate CPU usage during a period of moderate load and elasticity; a common occurrence in cloud deployments which should not stress an ideal cloud monitoring tool. In the following experiments, CPU usage refers to the over all percentage of resources within the cloud deployment which are devoted to performing monitoring. This includes all components performing monitoring: clients, servers and in the case of Varanus, all the constituent services.

Experiment 1

Figure 18a shows the CPU usage of Nagios and Varanus while the cloud deployment encounters a period of moderate, sustained demand and scales accordingly. CPU usage here in the overall percent of cpu time in the deployment which is devoted to providing monitoring functions. In the case of Nagios, this includes the monitoring servers, scripts running on monitored hosts and the backend

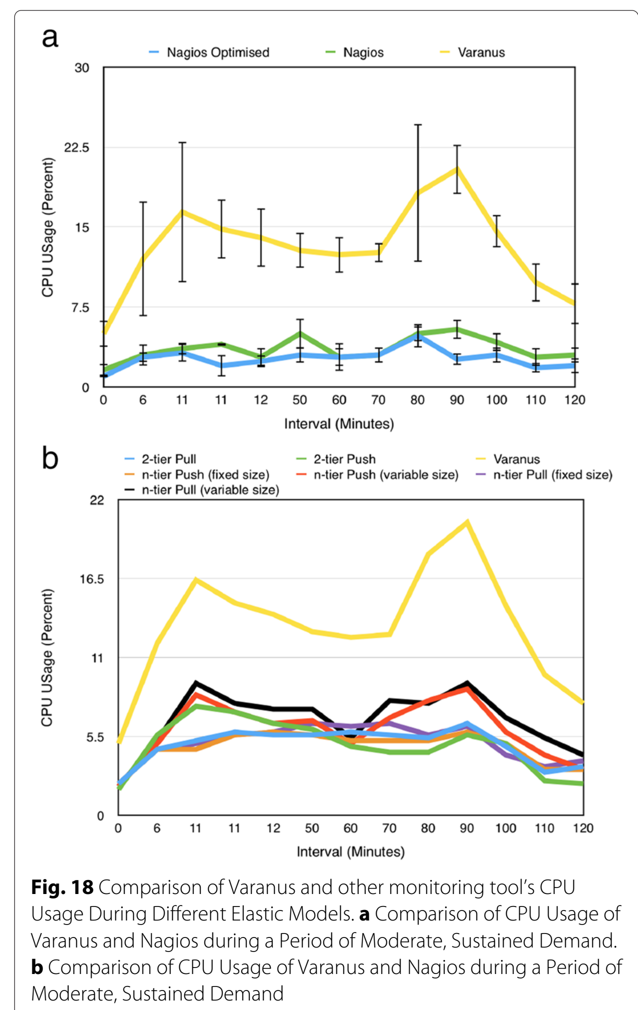

Fig. 18 Comparison of Varanus and other monitoring tool's CPU Usage During Different Elastic Models. **a** Comparison of CPU Usage of Varanus and Nagios during a Period of Moderate, Sustained Demand. **b** Comparison of CPU Usage of Varanus and Nagios during a Period of Moderate, Sustained Demand

database. In the case of Varanus, it is the percentage of CPU time used by all Varanus components. As Nagios follows a client-sever architecture it has a small number of dedicated servers and a large number of clients. Nagios, therefore heavily loads its servers and imparts a small load upon the clients. Varanus, is however decentralised and uses a small amount of CPU time on all participants and a larger percent on certain under-utilised hosts. Varanus, therefore uses a greater overall percentage of CPU use but amortises that cost over the entire deployment. This is evident from the range of values shown by the error bars of Varnus on Fig. 18a; as Varanus distributes the cost of monitoring and uses a resource aware approach its resource usage is highly variable dependant upon the available resources. Nagios meanwhile has a more conservative resource usage but heavily loads a small number of machines. As those machines must be provisioned in addition to the monitored deployment, they represent an additional overhead. Varanus, while more costly, attempts to use the free resources within that deployment and unless it is heavily loaded; does not require additional VMs. Additionally, with regards to the previous experiments, Varanus manages to achieve constant low latency monitoring with only double the resource usage of Nagios which has a very modest resource usage but can incur a very high monitoring latency.

Experiment 2

Figure 18b shows the CPU usage of Varanus and other monitoring architectures during a period of moderate demand. Again, Varanus due to its peer to peer, resource aware design consumes more resources than the other monitoring architectures. Worthy of note is the resource consumption of the two variable n-tier architectures which consume greater resources than the other monitoring architectures. As noted in previous experiments, these architectures have the lowest monitoring latency of set of evaluated architectures. They achieve lower latency by provisioning additional VMs to perform monitoring, this in turn results in a greater CPU usage. Varanus provisions no additional VMs (at least in this experiment) and achieves lower monitoring latency than the other monitoring architectures.

Evaluation summary

Our evaluation demonstrates that Varanus is able to perform low latency monitoring with conservative CPU usage even at scale. By utilising a decentralised architecture Varanus can propagate state and perform analysis without heavily loading individual VMs and by exploiting the plentiful low cost intra-cloud bandwidth of IaaS clouds Varanus can further reduce computation. As Varanus attempts to make use of existing resources, monitoring performance can suffer if resources are limited. This is the primary factor affecting Varanus performance and can be avoided by provisioning additional unloaded VMs.

When compared to Nagios, a diminishing yet still prolific monitoring tool, Varanus offers superior out of the box performance for cloud monitoring. Nagios can certainly be used effectively for cloud monitoring, however this requires significant modification backed by manpower and experience and used in conjunction with automation tools such as Puppet or Chef. When deployed in their default configuration or even with a number of patches, Nagios suffers from increasing monitoring latency and CPU usage proportional to the scale of the deployment.

Evaluation limitations

All the experiments detailed in this paper have been under controlled conditions with simulated load and artificial usage patterns. While these patterns have been deprived from real world trace data and are run on a genuine public cloud, they do not capture the full degree of spontaneity and variability which occurs under non-simulated conditions. Despite making significant inquiries, (quite understandable) no real stakeholders in large scale cloud systems were willing to test Varanus or participate in a manner which would yield publishable data. Despite the lack of real world testing the experimentation that has been performed has yielded results which should be consistent with what occurs under non simulated conditions. It would, however, have been preferable to include empirical evidence as to this in this paper.

Conclusion

Effective monitoring of large cloud systems is essential. Without appropriate monitoring anomalies, faults, performance degradation and all manner of other undesirable phenomenon can occur, unchecked. Previously it has been common practice for administrators to consume monitoring data and enact the appropriate changes to the system to ensure continued correct operation. With large cloud systems, this is no longer feasible. The scale, elasticity and complex of this class of systems prohibits a administrators from having a detailed holistic view of the system. Furthermore, many current tools are incapable of tolerating the scale and elasticity and deliver monitoring state in a non timely manner. Thus, what is required is an autonomic monitoring system which can collect and analyse monitoring data in a timely fashion and reduce the burden on human administrators. We have presented Varanus as this tool. By leveraging concepts from peer to peer, volunteer and highly decentralised computing we have designed Varanus to tolerate scale and elasticity and to act in a resource aware fashion. When evaluated against current tools, Varanus demonstrates its ability to maintain low latency monitoring with an acceptable resource overhead

which is amortised over the entire cloud deployment. We therefore contend that the design of Varanus is highly suitable for monitoring large scale systems and is a valid alternative to older monitoring schemes.

Endnotes

[1]named for the genus of the monitor lizard

[2]http://net.doit.wisc.edu/~dwcarder/rrdcache/

[3]https://lists.oetiker.ch/pipermail/rrd-developers/2006-August/001754.html

[4]The Quince Monitor Lizard (Varanus melinus) is of the genus Varanus

[5]http://puppetlabs.com

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
The core of this paper is based on work developed for JSW's PhD thesis at the University of St Andrews, supervised by AB. Both authors read and approved the final manuscript.

Acknowledgements
This research was supported by a Royal Society Industry Fellowship and an Amazon Web Services (AWS) grant.

Received: 4 March 2015 Accepted: 19 June 2015
Published online: 14 July 2015

References
1. Ward JS, Barker A Observing the clouds: a survey and taxonomy of cloud monitoring. J Cloud Comput 3(1):1–30
2. Ward JS, Barker A (2013) Varanus: In Situ Monitoring for Large Scale Cloud Systems. In: IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE. pp 341–344
3. Ward JS, Barker A (2014) Self managing monitoring for highly elastic large scale cloud deployments. In: Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing. DIDC '14. ACM. pp 3–10
4. Ward JS, Barker A (2012) Semantic based data collection for large scale cloud systems. In: Proceedings of the Fifth International Workshop on Data-Intensive Distributed Computing (DIDC). ACM. pp 13–22
5. Nagios. Nagios - The Industry Standard in IT Infrastructure Monitoring. http://www.nagios.org/
6. Massie ML, Chun BN, Culler DE (2004) The ganglia distributed monitoring system: design, implementation, and experience. Parallel Comput 30(7):817–840
7. Riemann. http://riemann.io/
8. Google (2014) Google Protocol Buffers. https://developers.google.com/protocol-buffers/docs/overview
9. Amazon CloudWatch. http://aws.amazon.com/cloudwatch/
10. Datta A, Sharma R (2011) GoDisco: selective gossip based dissemination of information in social community based overlays. In: Proceedings of the 12th International Conference on Distributed Computing and Networking. Springer-Verlag, Berlin, Heidelberg. pp 227–238. http://dl.acm.org/citation.cfm?id=1946143.1946163
11. Jelasity M, Montresor A, Babaoglu O (2005) Gossip-based aggregation in large dynamic networks. ACM Trans Comput Syst (TOCS) 23(3):219–252
12. Renesse RV, Minsky Y, Hayden M (1998) A gossip-style failure detection service. In: Middleware'98. Springer. pp 55–70
13. Dressler F (2006) Weighted probabilistic data dissemination (wpdd). Dept. of Computer Science
14. Ongaro D, Ousterhout J (2014) In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). USENIX Association, Philadelphia, PA. pp 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
15. Monnerat LR, Amorim CL (2006) D1ht: a distributed one hop hash table. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006, 20th International. IEEE. p 10
16. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H (2001) Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Comput Commun Rev 31(4):149–160
17. Voulgaris S, Gavidia D, Steen MV (2005) Cyclon: Inexpensive membership management for unstructured p2p overlays. J Netw Syst Manage 13(2):197–217
18. Gupta I, Birman K, Linga P, Demers A, Renesse RV (2003) Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In: Stoica I (ed). Peer-to-Peer Systems II. Springer Vol. 2735. pp 160–169
19. Hayashibara N (2004) Accrual failure detectors. PhD thesis, Citeseer
20. Oetiker T (2005) RRDtool. http://oss.oetiker.ch/rrdtool/
21. Stutzbach D, Rejaie R (2006) Understanding churn in peer-to-peer networks. In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. ACM. pp 189–202
22. NASA. NASA-HTTP Web Traces. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html
23. EPA Research Triangle Park, NC EPA-HTTP Web Traces. http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html
24. University of Saskatchewan University of Saskatchewan Web Traces. http://ita.ee.lbl.gov/html/contrib/Sask-HTTP.html
25. Clarknet Baltimore Clarknet-HTTP Web Traces. http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html
26. CopperEgg. http://copperegg.com/
27. Data Dog - Cloud Monitoring as a Service. https://www.datadoghq.com/

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com