Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

A design space for dynamic service level agreements in OpenStack

Craig A Lee[1*] and Alan F Sill[2]

Abstract

This paper does a systematic review of the possible design space for cloud-hosted applications that may have changing resource requirements that need to be supported through dynamic service level agreements (SLAs). The fundamental SLA functions are reviewed: Admission Control, Monitoring, SLA Evaluation, and SLA Enforcement – a classic autonomic control cycle. This is followed by an investigation into possible application requirements and SLA enforcement mechanisms. We then identify five basic Load Types that a dynamic SLA system must manage: Best Effort, Throttled, Load Migration, Preemption and Spare Capacity. The key to meeting application SLA requirements under changing surge conditions is to also manage the spare surge capacity. The use of this surge capacity could be managed by one of several identified load migration policies. A more detailed SLA architecture is presented that discusses specific SLA components. This is done in the context of the OpenStack since it is open source with a known architecture. Based on this SLA architecture, a research and development plan is presented wherein fundamental issues are identified that need to be resolved through research and experimentation. Based on successful outcomes, further developments are considered in the plan to produce a complete, end-to-end dynamic SLA capability. Executing on this plan will take significant resources and organization. The NSF Center for Cloud and Autonomic Computing is one possible avenue for pursuing these efforts. Given the growing importance of cloud performance management in the wider marketplace, the cloud community would be well-served to coordinate cloud SLA development across organizations such as the IEEE, Open Grid Forum, and the TeleManagement Forum.

Keywords: Service level agreements; Autonomic computing; Live migration; OpenStack

Introduction

Service level agreements (SLAs) are used to define the necessary Quality of Service (QoS) for an application or user in an IT system. SLAs originally were defined as a contractual document between IT resource providers and consumers that involved cost analysis and pricing, along with financial incentives or penalties. For performance-critical applications, though, such contractual SLAs are not sufficient. Performance-critical applications require SLAs whereby the computing infrastructure monitors, detects, and responds to changes in demand to ensure that application-level processing requirements are met. Furthermore, changes in demand may be caused not just by new applications being instantiated, but also by changes in demand by the running applications themselves. Some applications may have unpredictable changes in their processing demands and associated service levels. Even if changes in demand are somewhat predictable, it would still be desirable for the cloud service provider to be able to accommodate such changes without having to renegotiate a new SLA.

Hence, *dynamic* SLAs are required. This will be particularly necessary in computing cloud that are, by nature, multi-tenant environments where many applications may have changing service level requirements. What we want to avoid is forcing users to over-specify their service level requirements in order to satisfy future changes in their demand. If users were allowed to do so, then applications would simply acquire excess resource capacity and then let it sit idle the vast majority of the time. This would effectively *fragment* the cloud capacity and reduce the overall utilization.

By providing dynamic SLAs, we are attempting to satisfy competing goals: (a) ensure that every running application

*Correspondence: lee@aero.org
[1]Computer Systems Research Department, The Aerospace Corporation, P.O. Box 92957, El Segundo, CA 90009, USA
Full list of author information is available at the end of the article

component meets its deadlines, while (b) enabling the cloud scheduler to maximize resource utilization, thereby "doing more with less". By maximizing resource utilization, and understanding the possible aggregate surge requirements, it should be possible to do better overall capacity planning. That is to say, it should be possible to better determine the minimum amount of excessive surge capacity that needs to be available at any time. Doing so should help minimize the necessary overall cloud size, and reduce all associated costs, e.g., footprint, power, HVAC, staffing, etc.

Much work in cloud SLAs involves enforcing non-functional properties, such as compute node locality (zones), long-term storage preservation, and storage redundancy. For this white paper, however, we will focus just on performance metrics. Addressing non-functional properties will be addressed at a later time.

As noted already, the SLA mechanisms presented and discussed here will not be contractual in nature. That is to say, they will not involve two human organizations entering into an agreement for specific service levels between a provider and a consumer that carry penalties and rewards. We will also be considering performance management from both the consumer's and provider's perspective. We will not be considering one-sided goals, such as optimizing revenue. While many optimization problems, such as optimizing revenue [1], can be NP-hard requiring heuristic solutions, they do not address the performance requirements of individual user applications.

Hence, we will be developing technical, *machine-enforceable* SLA mechanisms, that a cloud provider can offer as a service, and a consumer can choose to use or not. These machine-enforceable mechanisms for dynamic SLAs will provide a probabilistic guarantee for performance. *The goal is to provide the user with a reasonable expectation that performance requirements will be met, through mechanisms that are reasonable for the provider to implement and support for multiple applications.*

In this paper, we begin by reviewing the fundamental functions necessary for SLAs and their enforcement. We then survey and investigate the possible design choices and implementation options. We conclude with a draft research and development for SLAs in OpenStack. While this particular R&D plan targets OpenStack as the test vehicle for planned work, any research results should be widely applicable to other cloud software stacks.

Fundamental SLA functions

Dynamic SLAs are actually an instance of an *autonomic control cycle: monitoring, analysis, planning, execution* – whereby systems can monitor themselves and maintain a target behavior [2]. In the context of dynamic SLAs, however, we will use the following four major functions:

- **Admission Control.** When a user wishes to instantiate a new application, the user must specify the required performance parameters for each of the application components to be instantiated. The cloud provider must then make a determination whether if sufficient capacity is available to adequately service the new application once started. An application component may consist of multiple servers that communicate in a specific topology. Hence, the cloud provider must determine if there are adequate cycles, memory space, disk space, and disk bandwidth for each application server, along with adequate network bandwidth among them. If there is, then the application can be started.
- **Monitoring – Metrics Collection.** While applications are running, the cloud infrastructure and the applications must be monitored. Monitoring must be as unobtrusive as possible, but must also capture essential data to determine if performance goals are being met. One or more monitoring systems could be used to collect data from different levels of the entire computing infrastructure. In a cloud environment, this could include monitoring the physical servers, hypervisors, the guest OSs, and the virtual applications themselves. While different monitoring systems could be used, all collected information must be collated and made available for the next functions.
- **SLA Evaluation.** Once an SLA has been established, the application has been started, and various performance metrics are being collected, there must be an agent that compares the SLA targets with the observed metrics, and determines when an application's performance has gone, or is going, "out of spec". For contractual SLAs, this could be termed an *SLA violation*, but for dynamic SLAs, this more accurately denotes that simply a threshold has been crossed requiring a response. A key issue for this agent is how to map the SLA metrics to the observable metrics. SLA metrics may be expressed in units that are meaningful at the application level and a *semantic gap* may exist between the metrics that are actually being collected.
- **SLA Enforcement – Violation Response.** An important issue for SLA Enforcement is whether an application's resource demands are expected to be static throughout its execution, or whether they can vary in a predictable or unpredictable manner. If an application's demands are expected to be relatively constant, then *static throttling* methods can be used. However, if an application's demands can vary, perhaps unpredictably, then it's behavior must be monitored to determine if it has gone "out of spec". For a performance-critical application, the primary goal of a machine-enforceable, dynamic SLA is to

pro-actively bring the application back into compliance. This requires some type of "control knobs" on the infrastructure or on the application itself.

SLA design options and approaches

Within each one of these fundamental SLA functions, there are further technical design issues that must be addressed. For each there are usually several implementation options with different challenges and trade-offs. We now put these fundamental functions into a general SLA architecture, as illustrated in Figure 1, and discuss them in more depth.

Admission control

Admission Control must maintain a *total cloud capacity document*, from which the currently available capacity can be derived or maintained. A total cloud capacity document must capture the capacity of all resources within a cloud data center, in addition to their network topology. Hence, in the most general case, all servers must be identified, along with their clock speeds, amounts of memory, local disk, and total network bandwidth. These resources must be placed in a network topology that includes routers and switches that have their own performance capabilities. Such network topologies are typically represented as a graph.

Likewise when a user submits an application for admission, the user must submit a service level request that captures the desired performance requirements that could be in a similar graph structure. Clearly for many applications, these resource graph structures may be simple, such as an individual server, or have a well-known, stereotypical structure. An example of this might be simply streaming a movie to a viewer. Some applications, however, may have detailed, highly structured, and even hierarchical SLA requests.

When the Admission Control agent receives a service request, it must consult the available capacity document and determine if the service request can be supported by the available capacity. When expressing network capacity and requirements as graph structures, this entails a *graph embedding* or *graph matching* problem. This is a well-known problem encountered in many other areas of computer science and IT. Timberwolf [3], for example, is a successful tool for doing die layout for integrated circuits consisting of millions of transistors and the wire runs that must connect them. Metis [4] is graph partitioning tool that might be applicable in this application domain. Also, a Network Calculus [5] could be used to provide a theoretical framework from which to model resource allocations and service guarantees. In computer networks, the concept of *service curves* are used that determines the relationship between arriving and departing network flows.

Admission control could, however, be greatly simplified by avoiding explicit graph presentations. The network switches connecting servers in a data center are often arranged in a hierarchy to achieve scalable bandwidth. That is to say, the switches are arranged in a *fat tree* where more bandwidth is available higher up in the tree, thereby avoiding *tree saturation* for network flows closer to the root, or backbone, switches. Under such conditions, the topology of the switch fabric becomes less important. Hence, we could ensure that the switch fabric does not become saturated simply by managing the aggregate BW demand in to and out of each of the servers. In this case, Admission Control only requires that each application

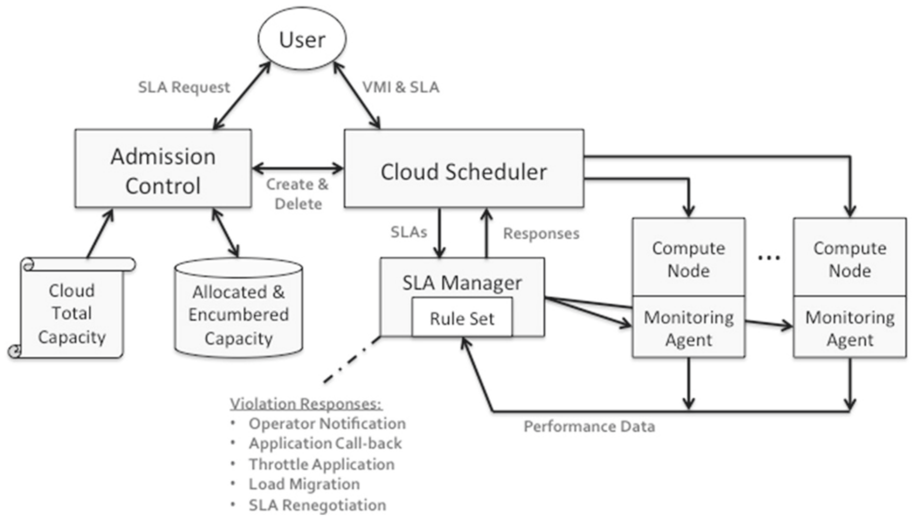

Figure 1 A General SLA Architecture.

process is allocated where there are sufficient cycles and network bandwidth at the network interfaces.

While this addresses the issue of network bandwidth, an application may also have network latency constraints between pairs of servers. In this case, then Admission Control must ensure that the servers are allocated "close" to one another by some metric, e.g., the number of network hops. While using a graph structure to represent sets of latency requirements is most general, latency requirements could be addressed by evaluating such a distance metric for simple pair-wise allocations.

In terms of existing standards, WS-Agreement [6] could be used to represent SLA service terms. WS-Agreement defines a domain-agnostic protocol whereby agreements can be established. This means that WS-Agreement can be used with different *term languages* as needed by the application domain. Hence, the challenge here is to define a term language that could capture the necessary application component performance characteristics, be they graph structures or simpler sums. WS-Agreement provides a formal definition for SLA representations and defines a single round of *offer-accept* message exchange. The provider offers an initial *template* of possible service terms. A service consumer replies with a completed template. The provider must then accept or reject the offer. There are a growing number of implementations for WS-Agreement, along with some valuable lessons learned [7].

Building on this, WS-Agreement Negotiation [8] defines a protocol for multiple rounds of negotiation, whereby a tree of offers and counter-offers is built. The tree root is the initial offer. Tree branches represent different sequences of offers and counter-offers, where service terms are adjusted in an effort to find mutually satisfactory terms.

It is actually very important that tools like WS-Agreement are domain-agnostic, since application domains can have widely different terms that are relevant to possible SLAs. Nae et al. [9], for example, present an interesting collection of SLA terminology and parameters for Massively Multiplayer Online Games that are of interest to providers and users. Given that many other domains will have similar but different sets of parameters, we can conclude that a flexible, pluggable architecture for SLA monitoring and reporting of performance metrics is highly preferable. While it might be possible to define a very basic SLA nomenclature, taxonomy and terminology, some degree of extension and customization will have to be allowed.

An outstanding issue here that we do not directly address in this paper is how to map high-level system requirements into lower-level metrics that can be measured and managed. Work has been done in this area, however. As part of the SLA@SOI project [10], the EVEREST Reasoning Engine translates SLA Abstract Syntax Objects into events in an Event Calculus [11]. This Event Calculus specifies patterns of events that should, or should not, occur within a specified period of time. The Detecting SLA Violation infrastructure (DeSVi) also enables mappings between SLA parameters and resource metrics by utilizing mapping rules with domain specific languages [12]. Similarly, the Quality Assurance for Distributed Services project (Qu4DS) manages the translation of SLA parameters by profiling the service provider [13]. Further work in managing the translation of application-oriented SLA requirements needs to be addressed as future work. For near-term experimental purposes, such translations can be managed by hand, since we will be focusing on the effectiveness of the SLA enforcement mechanisms themselves.

Another key challenge here is how to represent and manage the notion of *encumbered surge capacity* for dynamic SLAs. Many applications and application domains can be managed by static SLAs. A prime example here is simply streaming a video to a consumer. The consumer wants to watch the video with smooth performance, i.e., no pauses or drop-outs. The resource allocation necessary to accomplish this is known and constant. In multi-tenant clouds, however, with complex applications that will have varying computational needs, the challenge is how to dynamically meet these requirements within the context of an existing agreement, without having to resort to a heavy weight renegotiation and re-instantiation process. In a very real sense, a dynamic SLA is a quintessential use of the on-demand resources and flexibility offered by cloud computing.

This notion of encumbered surge capacity is also key for overall capacity management. Many dedicated systems are currently sized based on their expected *worst case* behavior rather than their *average case* behavior. This means there is dedicated excess capacity only to be used during worst case surge processing requirements. Having this dedicated excess capacity drives the overall system size, and total cost of ownership and operation, e.g., footprint, power, staffing, etc. One of the value propositions for cloud computing is the available of on-demand resources. That is to say, a cloud should be able to offer a joint pool of spare capacity to all tenant applications. Hence, a smaller pool of excess capacity should be needed, rather than having dedicated excess capacity for each application.

An alternative to maintaining surge capacity, however, is simply to allow resource *overbooking* [14]. An application's usage patterns for CPU, memory use, storage, and networking can be profiled and factored into the model for overbooking to achieve optimal packing of applications into a given set of virtualized resources. That is to say, the expected usage patterns can be used as an aid for the opportunistic packing of different types of applications (or differently-behaving instances of the same application) onto resources in order to achieve balanced overall utilization of the resources over all types of utilization within the application portfolio. As a simple example, I/O intensive applications might be booked with cpu-intensive ones that do not require a lot of I/O, to achieve an optimal overbooking. As reported in [14], though, a combination of modeling, monitoring, and prediction techniques must be used to avoid exceeding the total infrastructure capacity. In any case, application scheduling schemes should be sufficiently flexible to recognize performance degradation in a given metric that may come from overuse reported in another metric, and the SLA control scheme must be sufficiently powerful to allow for dynamic control over application types that can be mixed for optimal overall usage.

Once the cloud manager has determined that there is sufficient capacity to host the new application component – by whatever means used – the user can actually submit the component VMs, along with the SLA document, to the Cloud Scheduler. At this point, the Cloud Scheduler informs Admission Control that the components are actually being instantiated. This "inks the deal". The SLA is not in force until this point. The Scheduler starts the VMs on one or more Compute Nodes, and informs the SLA Manager of the new processes and the load that is expected. The SLA Manager may inform one or more Monitoring Agents on each Compute Node of metrics that need to be collected.

Monitoring

The actual monitoring could be done by a number of different existing tool sets. These include Ceilometer [15], Ganglia [16], Nagios [17], and Zenoss [18], to name a few. We will not review the details of these tools here. Rather, we identify key design alternatives that can determine the effectiveness and responsiveness of any SLA enforcement mechanism.

These design alternatives involve *what*, *where*, and *when* to monitor. The Monitoring Agents could monitor at different levels in the system stack on each Compute Node:

- *Host OS/Hypervisor.* Here the Monitoring Agent could capture all traditional operating system metrics at the hardware level, e.g., percentage CPU time per VM, memory usage, disk I/0, network I/0, etc.
- *Guest OS.* Monitoring here enables the Agent to collect operating system information specific to one VM.
- *Application Level.* As opposed to the previous two levels, this requires that the application be modified to provide a monitoring interface whereby application-level performance metrics can be obtained.

After this data is initially captured, it can be used at different times. All of the monitored data is archived in a database that can be queried by the SLA Manager. The Manager can periodically apply a rule set from the SLA information, provided by the Cloud Scheduler when VMs were instantiated, to determine if an SLA is being violated, or requires any corrective action. Given the amount of data in the database, the SLA Manager could also do *long-term trending analyses* that could not be identified from single metrics or events. However, rather than waiting for information to be deposited in the database, the SLA Manager could also set *stream triggers* as close to the meters themselves in the Monitoring Agents. These stream triggers are lightweight, state-less (or very low state) triggers that are easy to evaluate. Hence, they could provide the SLA Manager with an earlier notification of a potential problem.

These monitoring options, however, are all *reactive* – they only react to SLA violations after the fact. It is possible to do more *pro-active* SLA enforcement by monitoring further up in an application's processing chain. By doing so, the conditions that cause an SLA violation might be detectable before the violation actually occurs.

Consider that an application consists of an Input Buffer that partitions work units over a number of servers, as illustrated in Figure 2. After processing, these work units are collected by a Collector that monitors the how long it took each work unit to get processed after entering the Input Buffer. If work units are exceeding the application's latency requirement, additional servers could be spun-up to process more work units and reduce overall latency. This is reactive downstream monitoring since the SLA Manager can only decide to add more servers after latencies have exceeded some threshold. However, if the depth of the Input Buffer were monitored, then more pro-active SLA enforcement could be done, where more servers could be added when the buffer depth exceeds a threshold, but before the work unit latency actually starts to exceed application requirements.

Figure 2 also illustrates the fact that a large cloud data center may require and be composed of multiple SLA Managers to address issues of scalability. SLA Managers could be distributed across the cloud infrastructure, and also the applications themselves. These managers could communicate, or gossip, in a peer-to-peer fashion, be organized into hierarchies, or into other useful structures. This notion of distributed SLA managers implies that there must also be distributed SLA models by which overall system behavior, and individual application behavior, can be managed. The development and evaluation of such distributed SLA models is an outstanding goal.

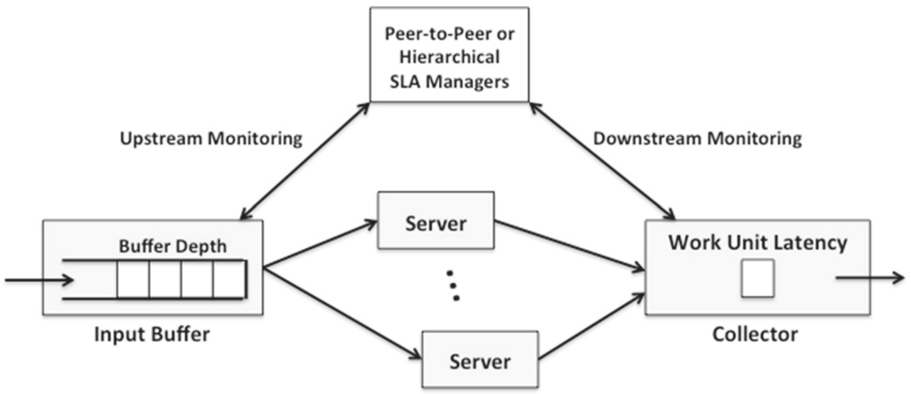

Figure 2 Distributed Monitoring of Different System Components.

SLA evaluation

Deciding when a server has crossed a load threshold is not a simple numerical comparison. A server's load at any one instant in time may fluctuate about a fixed threshold value. Basing decisions on each fluctuation could cause load migrations and VM instantiations that involve more overhead than the benefits realized. To prevent such behavior, more robust statistical methods are typically employed. Such methods include *Median Absolute Deviation, Interquartile Range,* and *Iterative Local Regression*.

In addition to dampening out some of the "high frequency" noise in system measurements, some amount of *hysteresis* should be built into SLA evaluations. This will prevent the SLA Manager from trashing between different enforcement mechanisms, such as load migration and load consolidation, in a cascading chain reaction.

We also note that if the surge probability of each application component is known, then the *joint probability* of any process on the server going into SLA violation could be determined. This joint probability could possibly be used to determine the best amount of spare capacity to maintain on a server, in order to minimize overall application impact due to load migration.

Developing such predictive models of system behavior is a common application of *machine learning* techniques. While machine learning has been a central topic in Artificial Intelligence for decades, the use of *reinforcement learning* in autonomic applications has been getting renewed attention [19]. A reinforcement learning system is essentially exposed to a sequence of *state-action pairs* to converge on a model of system behavior and optimal management policies. While reinforcement learning can be used "in the absence of explicit system models, with little or no domain-specific initial knowledge", *model-based* reinforcement learning can shorten the training process, but can also be constrained by the defined model. Thus, reinforcement learning is commonly recognized as having the following issues:

- The number of observed state-action pairs needed to converge on optimal policies may be huge,
- When in an initial training period, the results might be very poor, which can be very problematic for use on an operational system,
- Some number of *exploration* actions need to be taken that may produce sub-optimal results in the short-term, but enable better results in the long-term, and
- Real-world applications must not exhibit *incomplete observability* – that is to say, the RL system must be able to monitor all system metrics that are relevant to understanding and controlling the target system's behavior.

To address some of these concerns, hybrid reinforcement algorithms have been developed to shorten the training process and improve the overall quality of system management [20]. Reinforcement learning has also been recently applied to cloud computing [21]. In this work, a reinforcement learning algorithm was used to configure sets of VMs for their (1) number of virtual cpus, (2) number scheduler credits, and (3) the available memory capacity to optimize the response time and throughput when servicing a canonical, three-tiered web application. Efforts are also being made to commercialize this technology. Numenta's *Grok* architecture, for example, uses a bio-inspired *cortical* learning algorithm implemented using a modified Hadoop engine that is hosted on Amazon EC2 [22]. This provides *automated streaming analytics* that can be used for system behavior prediction and anomaly detection.

While such approaches to managine cloud-based SLAs are intriguing, much work needs to be done to establish how well this would work in practical applications. Such predictive modeling could be coupled with monitoring system behavior from widely different parts of the cloud and its applications. Learning algorithms could also be used to do on-the-fly load classification and comparison to established system models. The SLA mechanisms should also be able to deal with inaccurate or untruthful estimations of application requirements being provided by users [23]. Ultimately, the ability to serve both behavior prediction and anomaly detection is an advantage that should also be investigated.

SLA enforcement

Once it has been determined that an SLA in not in spec, what can the system do to bring it back into spec? We discuss some options here.

- **Notification or Call-back.** This is the simplest option. The SLA Manager simply notifies an operator who manually changes application parameters to reduce resource demand. While this is not a machine-enforceable mechanism, it will nonetheless be a practical alternative in many situations.
- **Throttling.** Throttling mechanisms exist in modern operating systems that can limit the amount of resources a process can consume. Under throttling, applications have no dynamic options and an SLA violation is not possible. This is suitable for applications that have very stable service level requirements, or where being throttled will not adversely affect application goals. (An example is processing a work load "over night" where there is significant lee-way in how long the processing actually takes.)
- **Load Migration.** If a server becomes overloaded by any metric, e.g., cpu load, memory usage, disk IO or network IO, this can be remedied by migrating some of the load to other servers with more available capacity. This can be done by either *process migration* or *live VM migration*. Clearly the overhead of either migration technique would limit how quickly service levels could be effectively re-established. Process migration may take less time since a smaller memory volume may have to be transferred, but migrating entire VMs allows sets of processes to be transferred. While migration can be done without interaction with the running application, the application is "down" during the migration. Hence, migration itself may precipitate, or contribute to, an SLA violation. Some clouds currently support *live VM migration*.
- **Acquiring Additional Resources On-Demand.** Rather than moving load, the offending application could be notified (through a call-back) that it needs to acquire more resources on demand. This requires that the application know how to incorporate more servers into its processing chain, i.e., how to partition its workload across more servers. That is to say, the application developer must design the application to be able to incorporate additional servers.
- **SLA Renegotiation.** As a final option, SLA renegotiation could be done. This is obviously a heavy-weight option of last choice since it could potentially entail restarting parts of the application on newly acquired resources.

While these are all options for managing SLAs, in this paper we are focusing on those mechanisms whereby dynamic SLAs can be managed. Users must have confidence that if they don't over-specify their average SLA requirements, the infrastructure will be able to gracefully respond to changes in requirements. This fundamental requirement affects how SLA must be defined, evaluated, and enforced. SLAs must be defined in a *term language* whereby the relevant terms and parameters can be expressed. These terms and parameters must be evaluated with regards to the available enforcement mechanisms.

For the SLA mechanisms that have been identified, we now can define the following *Server Load Types* that can be used by the Cloud Scheduler and SLA Manager to enforce SLAs (as illustrated in Figure 3):

Type I Service Loads are simply *Best Effort*, i.e., application VMs that are allocated using existing methods with no SLAs. As a practical matter, not all user applications may need or want an SLA, and the cloud provider may not want to require that all tenants must specify an SLA. These processes may change their processing, memory, disk and network demand whenever they want. All tenant processes on a particular server, however, but will be constrained by the physical capabilities of that server. If a server becomes resource-bound in any way (compute, memory, disk, or network), then all processes on that server will simply slow down, depending on their profile of resource demands.

Type II Server Loads are strictly throttled. As noted before, this is appropriate for those application processes that have known and stable resource requirements. By knowing those requirements, the cloud scheduler can co-locate such processes and maximize server utilization. Since each process is throttled, each is guaranteed a minimum level of service.

Type III Server Loads depend on live migration to remedy an SLA violation. By identifying those processes that can be migrated, the cloud scheduler can better manage server utilization. When a server exceeds a load threshold (by some metric), either a process or entire VM can be migrated to another server that has sufficient capacity to reduce the load on the local server. However, some processes or VMs may tolerate migration better than others.

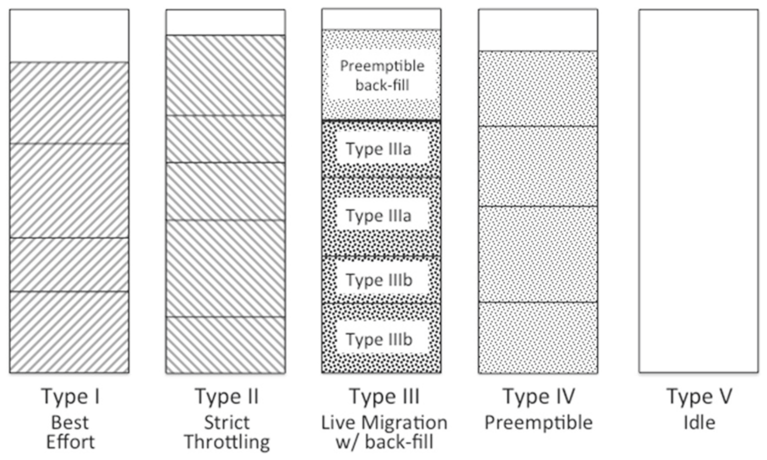

Figure 3 Server Load Types.

Since migration may take a non-trivial amount of time, migration itself may contribute to, or even cause, an SLA violation.

Hence, we can partition Type III Server Loads into **Type IIIa** and **Type IIIb Server Loads** that can, and can not, tolerate migration, respectively. For Type lIIb Server Loads that are less tolerant of live migration, a cloud scheduler could *delay* or *reduce* the number of necessary migrations by maintaining some amount of spare capacity on servers with migratable loads. The amount of this spare capacity could be managed to perhaps hit the "sweet spot" in the trade-off between wasting capacity and avoiding (or delaying) migration. Consider the policy whereby the cloud scheduler ensures there is enough spare capacity on a server such that any one application process can surge without triggering a migration. This can be expressed as:

$$\sum_{i=0}^{k-1} NormLoad_i + max_{0\leq i\leq k-1}(SurgeLoad_i) \leq MaxLoad$$

By ensuring that enough capacity is available for the maximum surge requirements of any process on that server, then any one process can surge without triggering migration. If multiple processes surge, then migration would be triggered at some time. (For this reason, it is important when application processes may surge for correlated reasons.) Of course, enough spare capacity could be maintained to enable $j < k$ processes to surge simultaneously, but this raises the amount of excess capacity that is maintained and its associated costs.

To limit the amount of excess capacity that is wasted, less than $max(SurgeLoad_i)$ spare capacity could be maintained. In this case, migration may not be avoidable for any one process, but at least migration would be delayed, depending on how long or how bad a surge is. The number of migrations over time might also be reduced.

For Type III SLAs (both a and b), it must be decided which process to migrate and to which target server. Such decisions must be codified as *migration policy*. Different migration policies can be defined that attempt to optimize different system or application metrics, sometimes defined as an *objective function*. For commercial operators that are concerned about power consumption and server utilization, this can be a monetized objective function based on the cost of power and the cost of SLA violations. This has been called the *Dynamic VM Consolidation* problem [24].

For performance-sensitive applications, however, migration policies could be based on a variety of different metrics to choose which process or VM to migrate:

- Fastest Migration Time (least time needed)
- Application Value (priority)
- Application Availability
- Maximum Load Reduction
- Load Reduction to Just Below Maximum
- Highest Correlation with Causing Excessive Load

Also, managing applications with non-stationary workloads with dynamic SLAs represents a fundamental autonomic control challenge. Given these applications' dynamic behavior, simple threshold comparisons will clearly be inappropriate. Methods will be needed for building *hysteresis* into the decision mechanisms. This has been investigated using *multi-size sliding window algorithms* whereby the *mean intermigration time* can be maximized [25].

It is clear that effectively using load migration for Type IIIa and IIIb SLAs will require extensive experimentation and experience. Which processes are appropriate for Type

IIIa and IIIb SLAs, which migration policies work the best, and how to manage their parameters, such as local surge capacity, are all open questions.

Rather than simply wasting the spare capacity to delay or reduce the number of live migrations, however, a similar technique to that used in [26] could be used to *back-fill* the spare capacity with *preemptible* processes or VMs. These are called **Type IV Server Loads**. Type IV Server Loads can be preempted without warning, and possibly restarted somewhere else at a later time, without significantly affecting the user's overall requirements. Thus, when a Type IIIb load goes into a surge, Type IV loads resident on the same physical server can be terminated, on-demand, to delay or reduce any necessary migrations.

Finally, **Type V Server Loads** simply represent spare capacity that is available on-demand, for both live migrations and for new VM instantiations. The amount of this spare capacity depends on the aggregate surge requirements of all applications.

Overall, the goal is to ensure that all application processes can meet their processing requirements, while minimizing the resources that must be maintained. This means strictly managing the excess capacity that must be maintained for surge requirements. It also means managing the spare capacity that is fragmented across all servers that are not fully loaded, and the total spare capacity available within the cloud as a whole. Theoretically, server load could be managed completely dynamically by live migration and on-demand instantiation, but stating Service Levels gives the cloud scheduler valuable information concerning the expected demands. Ideally this should enable the cloud scheduler to minimize the number of live migrations and instantiations that are necessary.

An SLA research and development plan

Having reviewed the fundamental SLA functions, and identified the available SLA enforcement mechanisms for OpenStack, along with the resulting Load Types, we now put all of this together into an architecture and a development plan.

An SLA architecture for OpenStack

Figure 4 presents an integrated SLA architecture for OpenStack. The User begins by sending an SLA request to the Nova-AC (Admission Control) service, using the standard WS-Agreement format. When Nova-AC is booted, it is initialized with the Total Capacity Document for this cloud. This document specifies the cloud's current total capacity, e.g., number and type of servers, amount of storage, network bandwidth, etc. The exact format of this document needs to be defined, but presumably it could follow the *SLA template* that the Nova-AC provides to potential users as part of the WS-Agreement process.

Based on the incoming SLA request, Nova-AC consults its database of allocated and available capacity. According to the discussion in Section 'Admission control', an admission control decision could be made simply by identifying a host where there is sufficient cycles and network bandwidth available, according to the requested Load Type (without having to maintain more complicated, graphical representations). If multiple rounds of negotiation are required, the WS-Agreement Negotiation [8] standard defines how this can be done.

Once the User essentially has an "SLA offer", the Virtual Machine Image and the SLA are submitted through the Nova-API to the Nova-Scheduler. Like all OpenStack services, the Nova-API is designed using a configurable

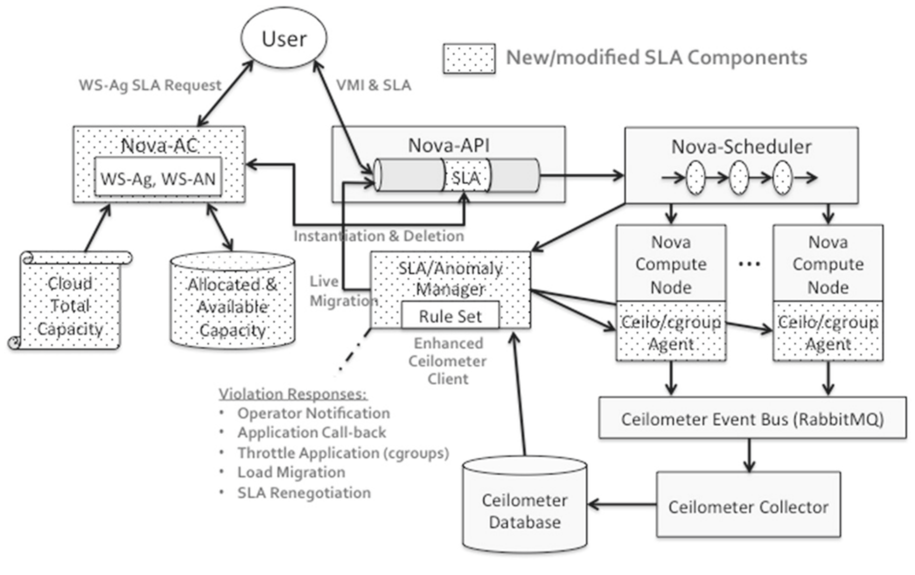

Figure 4 An SLA Architecture for OpenStack.

command pipeline. When booted, the command pipeline can be configured with different pipeline stages to include or omit different functionalities. To support SLAs, a new SLA pipeline stage will be included. When any operation involving an SLA is encountered, Nova-API will inform Nova-AC of the change in resources being allocated or released. It will likewise inform the SLA Manager of the same changes.

When VMs are actually being allocated, the Nova-Scheduler uses a *Filter and Weight* approach, as illustrated in Figure 5. To choose the host on which to allocate a VM, Nova-Scheduler first applies a set of filters to identify which hosts are possible candidates. A set of weighting functions are then applied to rank the candidates and identify the best one. This design allows OpenStack to support many different scheduling paradigms, depending on the filters and weighting functions available. OpenStack currently support several basic scheduling algorithms, e.g., random placement, random placement within an availability zone, and placement on the least loaded host. To support SLAs, additional SLA Filtering and Weighting functions will need to be written that makes the correct selections based on the SLA Load Type and capacity requested.

Once instantiated on a host compute node, a monitoring agent will have to be used to acquire the necessary performance information. Ceilometer is the OpenStack monitoring service under development. This uses the same RabbitMQ [27] instance used by Nova to collect all performance information and deposit it in a database. Other monitoring tools, besides Ceilometer, could be used. Zenoss, for instance, is much more mature than Ceilometer and offers many *ZenPacks* that can be installed to monitor many different parts of a system software stack, e.g., Apache Tomcat servers, PostgresSQL databases, and Java SNMP, just to name a few examples. Nonetheless, for initial development and evaluation purpose, Ceilometer could be used completely adequate.

Based on the Load Type requested, the SLA Manager will look for specific performance metrics from different host servers. For Type II throttled work loads, a mechanism, such as *cgroups* [28] in Linux, could be used. Linux cgroups allow hierarchical *control groups* to be defined. Each control group is associated with a limit on the amount of resources that can be consumed on that server, e.g., the percentage of cpu time, memory, disk IO, and network IO. These limits are actually enforced by tools such as Linux *CPUsets* [29]. User processes are assigned to different control groups based on the limit of resources they are allowed to consume. The operating system then enforces those limits when scheduling a process to run. To use this mechanism, the SLA Manager will have to manage the available cgroups on a specific servers, and which VM processes are assigned to them.

For Type IIIa and IIIb work loads, the SLA Manager will need to employ live migration to enforce policy. Live VM migration can generally be done transparently to the running applications. However, the time required for a live migration depends on the amount of memory currently in use by a VM, *Mem*, the number of open file descriptors (i.e., open files and network connections), *nOpenConn*, and the available network bandwidth, *BW*, between the current host and the migration target host.

$$T_{migration} = O(nOpenConn + (Mem/BW))$$

Further refinements to this relationship can be made. The performance costs of live migration can also depend on the internal details of the cloud implementation, such as the type of hypervisor used, as well as the storage and memory architectures [30]. A truly flexible SLA control

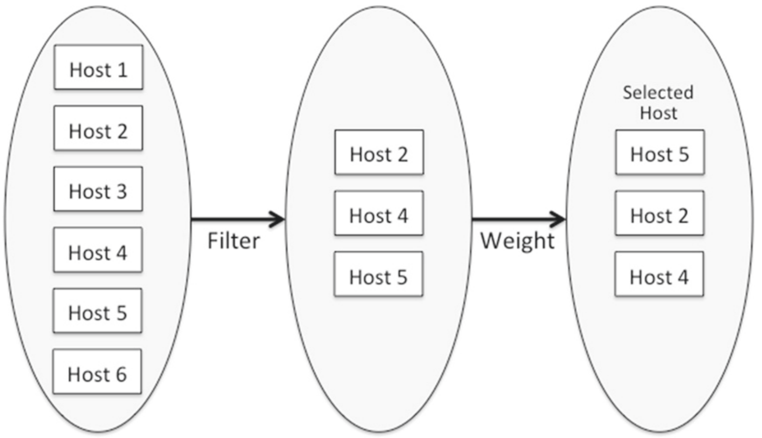

Figure 5 The Nova-Scheduler Filter and Weight Design.

architecture will allow for selection of the application parameters for which the cost model should be tuned.

Even if live migration is otherwise transparent to the application, this time delay may precipitate, or contribute to, an SLA violation. This underscores the importance of accurately evaluating $T_{migration}$ that applications may encounter, and how it may affect their SLAs. It may also be the case that CPU-intensive applications may be easier to migrate than ones that are highly inter-connected. Hence, a key goal will be to develop an accurate and predictive model of live migration that takes in all of these possible factors, and can be used to support different migration policies.

Finally we note that using the SLA architecture described here to effectively manage cloud resources will require a certain *population* of Load Types. That is to say, processes and VMs of each Load Type must be represented in some number and distribution whereby a cloud scheduler can effectively use them to manage the overall cloud utilization, while ensuring that individual application performance requirements are being met. This will be especially true for Load Types III and IV where back-fill scheduling is used to improve utilization. Experimentally determining workable Load Type populations will only be possible once there is a significant SLA architecture in place.

A research and development plan

With these design options and goals in mind, we now present a draft development and test plan. Clearly this SLA system will depend on Linux cgroups and live migration in OpenStack. cgroups are an established capability, so that is considered to be low risk. Live migration in OpenStack, however, is still maturing.

Live migration has been demonstrated in the OpenStack Folsom release, using KVM with libvert on ubuntu 10.09 or 12.04. Live migration using Xen has also been reported. The initial use for live migration in OpenStack, however, is not load balancing or SLA enforcement. Rather it will be used for *VM evacuation*. This is for basic server maintenance purposes, where all running VMs can be moved off of a server to allow software upgrades, hardware replacement, or other routine maintenance functions to be done. Server evacuation is currently intended to be manually managed by cloud administrators.

Given that the basic live migration capability has been demonstrated in OpenStack (since Folsom) for manual maintenance purposes, it should be possible to add the SLA "intelligence" whereby live migration can be used for performance management. Hence, we defined the sequence of tasks for the Research and Development Plan given below. This plan starts by building and evaluating just the core SLA enforcement mechanisms (Tasks 1 and 2). We must first show that these mechanisms are effective in providing reasonable guarantees of application performance. Once that is established, we can then consider building the rest of the supporting SLA management tools, i.e., Tasks 3, 4, 5 and 6. Proposing specific schedule milestones and budget are outside the scope of this paper.

A Research and Development Plan

1. **Build Core Enforcement Infrastructure**
   - Install Ceilometer on existing OpenStack cloud
   - Build a basic SLA Manager by enhancing the Ceilometer client
     - Periodically queries database for specific metrics
     - Applies rule sets to detect specific performance conditions on a per application basis
   - Demonstrate process throttling with cgroups
   - Demonstrate processes and VMs live migration
2. **Demonstrate SLA Manager Capabilities**
   - Build/construct synthetic/manufactured work loads
     - Provide work loads programmed to go through variable changes in demand
     - Build work load scenarios relevant to various application domains
   - Develop and evaluate migration policies
     - Assess migration parameters: spare capacity, migration time, load reduction/correlation, etc.
     - Evaluate how well live migration can be managed (inter-migration time)
     - Develop accurate, predictive model(s) of migration overhead
     - Compare process migration vs. VM migration
   - Evaluate overhead, responsiveness, and stability
     - How much overhead does monitoring incur to produce useful information
     - Evaluate how quickly can the system respond to and maintain SLA targets
     - Evaluate responsiveness vs. stability trade-off
3. **Develop Nova-AC Service**
   - Develop common WS-Agreement term language

- Develop Total Cloud Capacity Document
- Develop Allocated Capacity Document
- Investigate semantic mapping from application-level requirements to infrastructure-level metrics

4. **Develop Nova-API Pipeline Stage**
    - Establish interaction with Nova-AC and SLA Manager when instantiating application VMs
        - Enable accurate tracking of allocated and available capacity
        - Enable live migration managed by SLA Manager
5. **Develop Nova-Scheduler SLA Functions**
    - Develop the filtering and weighting functions to identify the best host for relevant Load Types
6. **Demonstrate End-to-End Integrations**
    - Develop or acquire non-trivial “operational-like” work loads
    - Manage SLAs for multiple apps simultaneously
    - Demonstrate that individual application requirements can be met while managing overall cloud requirements, e.g., utilization
    - Evaluate minimal population of Load Types (number and distribution) to effectively manage overall cloud resources
7. **Develop and Evaluate Learning Algorithms**
    - Identify all relevant system metrics necessary understand and control system behavior
    - Evaluate the trade-off between the length of training period versus eventual effectiveness
    - Develop and evaluate methods for enabling exploration actions that do not adversely affect operational systems
8. **Develop and Evaluate Distributed SLA Managers**
    - Evaluate P2P and hierarchical organizations
    - Develop and evalaute distributed SLA models, ultimately including learning algorithms
    - Evaluate overhead, responsiveness, stability of distributed SLA methods

Summary and recommendations

This paper has reviewed the basic requirements for providing dynamic service-level agreements, and developed a draft plan for implementing and evaluating such dynamic SLAs in OpenStack. This SLA architecture does require that applications understand what their own resource requirements are. For some existing applications, this information may be difficult to acquire. When an application is deployed on dedicated hardware, there may have been no provision for determining the actual requirements of each application component. As long as an application component never became an egregious bottleneck, everything was fine.

The proposed SLA architecture could, in fact, be used to determine a application’s actual requirements. Without specifying or enforcing SLAs in a separate test environment, the monitoring infrastructure could simply catalog the application’s behavior over time and various “operational” conditions. Once known, the application could be moved to an SLA-controlled infrastructure with appropriate SLAs in force. In any case, any further work should leverage other relevant projects in the overall cloud marketplace. The NSF Center for Cloud and Autonomic Computing has a number of projects concerning cloud SLAs [31].

The European Union has reported the results of a number of research projects, ranging from business-level SLAs to scientific SLAs [32]. These include the OPTIMIS [33], CONTRAIL [34], and SLA@SOI [10] projects. Given the wide interest in SLAs and the recognition that SLAs will be critical for a wide segment of the cloud marketplace, the TeleManagement Forum (TMF) has started an SLA working group [35] to develop industry best practices and standards, using the OGF WS-Agreement and WS-Agreement Negotiation as a starting point [36]. With these developments, the OpenStack community might eventually incorporate support for some form of dynamic SLAs.

Acknowledgments
The authors gratefully acknowledge the many useful comments from the organizers and participants of the *3rd International Workshop on Intelligent Techniques and Architectures for Autonomic Clouds (ITAAC 2013)* that improved the scope and completeness of this paper.

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

Authors’ contributions
CL produced an earlier version of this paper that laid out the fundamental capabilities necessary for dynamic service level agreements, identified possible control mechanisms based on the load types, and cast this as potential augmentations to the OpenStack service architecture. AS greatly improved this earlier version by clarifying and emphazing key concepts, identifying highly relevant citations, and fleshing out the research and development plan, as relevant to the new NSF Center for Cloud and Autonomic Computing at Texas Tech University. The authors read and approved the final manuscript.

Author details
[1]Computer Systems Research Department, The Aerospace Corporation, P.O. Box 92957, El Segundo, CA 90009, USA. [2]High Performance Computing Center, Texas Tech University, Lubbock, TX 79409, USA.

Received: 25 January 2014 Accepted: 4 June 2014
Published online: 07 November 2014

References

1. Casalicchio E, Menascé DA, Aldhalaan A (2013) Autonomic resource provisioning in cloud systems with availability goals. In: Proceedings of the 2013 ACM cloud and autonomic computing conference. CAC '13. ACM, New York. pp 1:1–1:10
2. Kephart JO, Chess DM (2003) The vision of autonomic computing. IEEE Comput 36(1):41–52
3. Sechen C, Sangiovanni-Vincentelli A (1985) The Timberwolf placement and routing package. IEEE J Solid-State Circuits 20(2):510–522
4. Karypis G METIS: Serial Graph Partitioning and Fill-Reducing Matrix Ordering. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
5. Jiang Y, Liu Y (2009) Stochastic network calculus. Springer, London, 2008. doi:10.1007/978-1-84800-127-5, ISBN 978- 1-84800-126-8
6. Waeldrich O, Battré D, Brazier F, Clark K, Oey M, Papaspyrou A, Wieder P, Ziegler W (2011) Web services agreement negotiation, Version 1.0. OGF GFD-R-P.193. http://www.ogf.org/documents/GFD.193.pdf
7. Battre D, Hovestadt M, Wäldrich O (2010) Lessons learned from implementing WS-agreement. In: Wieder P etal. (ed). Grids and service-oriented architectures for service level agreements. Springer
8. Andrieux A, Czajkowski K, Dan A Keahey K, Ludwig H, Nakata T, Pruyne J, Rofrano J, Tuecke S, Xu M (2011) Web services agreement specification (WS-Agreement). OGF GFD-R.192. http://www.ogf.org/documents/GFD.192.pdf
9. Nae V, Prodan R, Iosup A (2013) Autonomic operation of massively multiplayer online games in clouds. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing conference. CAC '13. ACM, New York. pp 10:1–10:10
10. The SLA@SOI Project. http://www.sla-at-soi.eu
11. Mahbub K, Spanoudakis G, Tsigkritis T (2011) Translation of SLAs into monitoring specifications. In: Wieder P, Butler JM, Theilmann W, Yahyapour R (eds). Service level agreements for cloud computing. Springer
12. Emeakaroha VC, Netto MAS, Calheiros RN, Brandic I, Buyya R, De Rose CAF (2011) Towards autonomic detection of SLA violations in cloud infrastructures. Future Generat Comput Syst 28(7):1017–1029. doi:10.1016/j.future.2011.08.018
13. Freitas AL, Parlavantzas N, Pazat J-L (2012) An integrated approach for specifying and enforcing slas for cloud services. In: 5th IEEE intl. conf. on cloud computing. IEEE
14. Tomás L, Tordsson J (2013) Improving cloud infrastructure utilization through overbooking. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing conference. CAC '13. ACM, New York. pp 5:1–5:10
15. The OpenStack Foundation. Welcome to the Ceilometer Developer Documentation! http://docs.openstack.org/developer/ceilometer/
16. The Ganglia Monitoring System. http://ganglia.sourceforge.net
17. Nagios Enterprises. Nagios. http://www.nagios.org
18. Zenoss Inc. Zenoss. http://www.zenoss.com
19. G Tesauro (2007) Reinforcement learning in autonomic computing: a manifesto and case studies. IEEE Internet Comput 11(1):22–30
20. Z Wang, X Qiu, T Wang (2012) A hybrid reinforcement learning algorithm for policy-based autonomic management. In: 9th International Conference on Service Systems and Service Management (ICSSSM). IEEE
21. C-Z Xu, J Rao, X Bu (2012) URL: A unified reinforcement learning approach for autonomic cloud management. J Parallel Distrib Comput 72(2):95–105
22. Ahmad S (2012) Automated machine learning for autonomic computing. In: 9th ACM International Conference on Autonomic Computing (ICAC). New York
23. Mashayekhy L, Nejad MM, Grosu D (2013) A truthful approximation mechanism for autonomic virtual machine provisioning and allocation in clouds. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing conference. CAC '13. ACM, New York. pp 9:1–9:10
24. Beloglazov A, Buyya R (2012) Optimal online deterministic algorithms and adaptice heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. Concurrency Comput Pract Ex 24:1397–1420
25. Belograz A, Buyya R (2013) Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. IEEE Trans Parallel Distr Syst 24(7):1366–1379
26. Marshall P, Keahey K, Freeman T (2011) Improving utilization of infrastructure clouds. In: CCGRID '11 Proceedings of the 2011 11th IEEE/ACM international symposium on cluster, cloud and grid computing. pp 205–214
27. GoPivotal, Inc. RabbitMQ. http://www.rabbitmq.com
28. Menage P, Jackson P, Lameter C (2006) CGroups. http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt
29. Derr S, Jackson P, Lameter C, Menage P, Seto H (2006) CPUSets. http://www.kernel.org/doc/Documentation/cgroups/cpusets.txt
30. Hu W, Hicks A, Zhang L, Dow EM, Soni V, Jiang H, Bull R, Matthews JN (2013) A quantitative study of virtual machine live migration. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing conference. CAC '13. ACM, New York. pp 11:1–11:10
31. The NSF Center for Cloud and Autonomic Computing. http://www.nsfcac.org
32. Kyriazis D (ed) (2013) Cloud computing service level agreements: exploitation of research results. http://ec.europa.eu/digital-agenda/en/news/cloud-computing-service-level-agreements-exploitation-research-results
33. The OPTIMUS Project. http://www.optimis-project.eu
34. The CONTRAIL Project. http://www.contrail-project.eu
35. The TeleManagement Forum. SLA Management. http://www.tmforum.org/SLAManagement/1690/home
36. (2013) The TeleManagement Forum. Enabling End-to-End Cloud SLA Management, TR 178, Version 0.7. http://www.tmforum.org/TechnicalReports/TR178EnablingEndtoEnd/50148/article.html

doi:10.1186/s13677-014-0017-1
Cite this article as: Lee and Sill: **A design space for dynamic service level agreements in OpenStack.** *Journal of Cloud Computing: Advances, Systems and Applications* 2014 **3**:17.