Admission Control of API Requests in OpenStack

Tatsuma Matsuki, Noboru Iwamatsu
Software Laboratory, Fujitsu Laboratories Ltd,
4-1-1, Kamikodanaka, Nakahara-ku, Kawasaki, Japan.
{matsuki.tatsuma, n_iwamatsu}@jp.fujitsu.com

***Abstract*—IaaS clouds have attracted much attention, and OpenStack has become a de facto standard software for building open-source IaaS clouds. However, the performance and scalability of OpenStack services still have room for improvement. This study focused on the admission control of API requests to improve performance. We revealed the basic effect of admission control by setting the concurrent connection limit of API requests in OpenStack, and we confirmed that admission control could improve the performance of OpenStack services by more than 50% in an overloaded situation. We also proposed a heuristic algorithm that adaptively tuned the concurrent connection limit by monitoring the statistics of the completed API requests obtained from a proxy server. Our experimental evaluation revealed that our algorithm helped avoid severe performance degradation in OpenStack.**

I. Introduction

Many applications and services have been delivered by using Infrastructure as a Service (IaaS) clouds. From their agility and scalability, IaaS clouds have greatly contributed to continuously deploy applications and systems. These deployments are accelerated by opening the cloud service APIs [1], [2]. Automated deployment tools, such as OpenStack Heat [3] and Ansible [4], utilize these APIs to execute the deployments. OpenStack Heat enables auto scaling of applications running on virtual machines (VMs) on IaaS by repeatedly creating and deleting VMs based on their workload. These operations involve frequent API requests. In addition, the recent Platform as a Service (PaaS) clouds, such as Cloud Foundry [5], are often built on IaaS clouds to achieve scalability and flexible management. A large-scale Cloud Foundry deployment is an example that leads to build more than one hundred of VMs on IaaS [6]. From these backgrounds, the performance (e.g., response times and throughput) is quite important for the recent IaaS clouds.

OpenStack [7] is the most well-known open-source software for building private and public IaaS clouds. It has the largest population as an open-source cloud in terms of community participants, developers and their activities [8]. Recent studies also focused on clouds using OpenStack [9], [10], [11] and some evaluated the performance of OpenStack [12], [13], [14]. However, research for improving the quality of service (QoS) of the OpenStack *services*, such as creating and migrating VMs, attaching volumes and virtual networks to VMs, is not sufficient. The QoS of the OpenStack service must be investigated to support large bulk API requests.

This paper focuses on admission control of the OpenStack API requests to improve the QoS in OpenStack. Two reasons motivate this work: 1) all OpenStack user requests originate from the API requests and 2) we can apply an admission controller on the proxy server without modifications in OpenStack codes. The latter is important because OpenStack is still being actively developed and a large release of OpenStack is continuously applied once every 6 months. We built a prototype of the admission controller running with HAProxy [15], which is often used as a load balancer in the OpenStack high availability (HA) configuration [16].

Admission control is a well-known approach to control the QoS of Web services and numerous existing studies have focused on this approach [17], [18], [19], [20], [21]. However, these existing studies are insufficient from two perspectives: 1) the effect and efficiency of admission control on the performance of IaaS remain unclear, and 2) the admission control that achieves high adaptability to IaaS workload is not investigated. We then applied an admission control schema that sets limits on the maximum concurrent connections (i.e., API requests) in OpenStack services. We synthetically generated workload on IaaS by using OpenStack Rally [22], which can generate workload following various kinds of common scenarios of OpenStack clients using APIs in OpenStack.

We also present an adaptive admission control algorithm that takes the IaaS workload characteristics into consideration and achieves high adaptability to various kinds of cloud configurations. Unlike existing studies, our approach could be applied without any reference performance, such as service level agreement (SLA), and preliminary benchmarking. Our main contributions made in this paper are:

1) We built OpenStack cloud environments with HA configuration, executed the benchmark of a VM booting scenario, and revealed the effect of admission control on the performance of OpenStack services, which was based on limiting the maximum concurrent connections.
2) We also proposed a heuristic admission control algorithm in OpenStack, which automatically tunes the concurrent connection limit based on the statistics obtained from the proxy server. We designed the algorithm to achieve high adaptability to various kinds of cloud environments and the characteristics of the workload generated on IaaS, especially on OpenStack.
3) We implemented a prototype of the admission control algorithm, which runs alongside the HAProxy. We confirmed that our algorithm achieves good performance of OpenStack services via experimental evaluation with two different types of cloud environments.

978-3-901882-89-0 @2017 IFIP

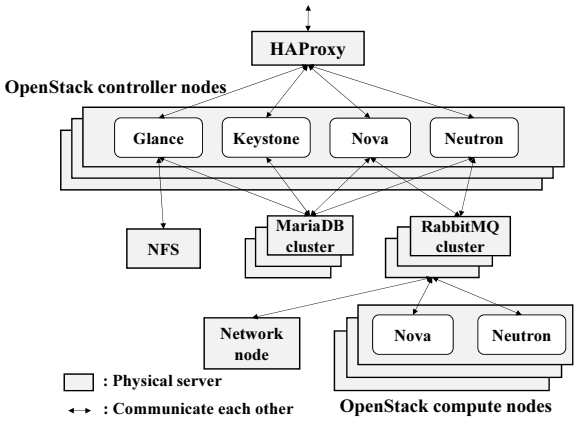


Fig. 1. OpenStack architecture and example of an OpenStack physical configuration. OpenStack controller nodes and individual components are configured to be redundant and scalable.

The rest of this paper is organized as follows. We introduce OpenStack architecture in Section II and explain admission control in OpenStack and its effect in Section III. Section IV proposes an adaptive admission control algorithm and Section V discusses its applicability. Section VI introduces some related studies. Section VII concludes and describes future works.

II. OpenStack Architecture

OpenStack [7] is a set of open-source software to build and orchestrate IaaS clouds. OpenStack consists of several key components called *projects*. In this paper, we summarize some of important projects and services provided by the projects.

- **Nova** provides VM management services such as VM creation, deletion and migration. Some API requests toward Nova service involve RPCs between the controller and compute nodes. The compute nodes are responsible for hosting VMs on the top of a hypervisor.
- **Neutron** provides a networking service, which enables VMs to connect the virtual and external networks.
- **Keystone** provides authentication for OpenStack users and services, endpoints discovery of OpenStack services.
- **Glance** is responsible for the management of VM images used for creating VMs.

These services provided by individual projects are available through REST APIs. These projects are loosely coupled and communicate with each other via the REST APIs. OpenStack users who want to build application on OpenStack also use these APIs via OpenStack client software.

A. Configuration Example

Fig. 1 illustrates OpenStack architecture and an example of physical configuration. OpenStack controller nodes in Fig. 1 host each OpenStack service described above. Each service in the controller nodes utilizes stateful backend nodes, which includes Network File System (NFS) cluster, MariaDB cluster called Galera [23]. The controller nodes communicate with OpenStack compute nodes via messaging service, which is often implemented with RabbitMQ [24], to assign tasks on the OpenStack compute nodes. The compute nodes are responsible for running VMs on the top of hypervisor such as KVM.

The configuration shown in Fig. 1 is an example of HA configuration in which all components are duplicated to make OpenStack services scalable and tolerant to faults. HAProxy [15] is used for load balancing of OpenStack controller nodes at the top of Fig. 1. HAProxy distributes the incoming HTTP requests to OpenStack controller nodes with a specified balancing discipline such as round robin and least connection. When the HAProxy balances the load of HTTP requests, it also routes the incoming request to each OpenStack service based on the port number of individual HTTP requests.

B. API Workflow Example

API requests for OpenStack services lead to pre-defined operations in OpenStack. We briefly introduce the example operations caused by a *boot server* API request in Nova service, which is used for creating and booting VMs. We focus on this *boot server* API requests to evaluate admission control in OpenStack, because it is the most basic service in IaaS.

When an OpenStack user wants to boot a VM, he/she issues a HTTP POST request to Nova service. After the Nova service accepts the request, the following operations are performed by OpenStack: 1) checking whether the token on the request is valid by using Keystone service, 2) accessing the database to obtain information required for booting a new VM and creating an entry for the new VM, 3) responding to this API request, 4) selecting an appropriate compute host to boot a VM by interacting with the database, 5) issuing a *boot server* task to the appropriate compute node via the messaging service and the compute node executes boot a VM, which includes fetching information of VM image using Glance service. These operation can be followed by some additional operations such as the network connection by using Neutron service.

III. Admission Control in OpenStack

All OpenStack services originate from the corresponding API requests. Therefore, the API requests should be appropriately controlled to prevent OpenStack from being overloaded and severe performance degradation.

A. Concurrent Connection Limit

Admission control of HTTP requests in Web services has been extensively investigated. Limiting the maximum concurrent connections (API requests) is a well-known approach to avoid overloading in Web services. We focus on the limitation of concurrent connections in the OpenStack services and implement the limitation at HAProxy as illustrated in Fig. 2. Fig. 2 depicts the HAProxy and backend OpenStack controller nodes, in which the concurrent connections are limited within l_n for OpenStack controller node n. To implement the limit, we use the *maxconn* parameter on HAProxy that can be set for individual backend servers. The limitation is also set for each OpenStack service such as Nova and Keystone. Each service possesses different limitations. The total concurrent

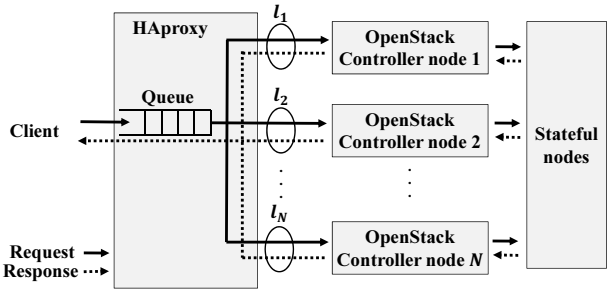


Fig. 2. Limiting concurrent connection on HAProxy

connection limit for a service is calculated as $L = \sum_n l_n$. When the number of concurrent connections exceeds the limit L, newly incoming requests are queued on HAProxy to wait for one of early arrival API requests to complete. By setting the limitation, the admission of API requests can be controlled when the large number of API requests arrives during a short time or the early arriving requests suffer from the delay. Note that we apply the limitation only to the requests issued by external clients, which means that the requests issued among OpenStack services are not regulated. In this study, the load balancing among OpenStack controller nodes is conducted with the least connection discipline.

In this work, we first investigated the effects of concurrent connection limit by experimentally building two different types of OpenStack IaaS cloud environments, which will be explained in the following subsection.

B. Experimental Setup

1) OpenStack configuration: We experimentally built two OpenStack cloud environments with two different sets of physical servers. All servers in the first server set denoted as *Cloud-A* have Intel Xeon E-2643 3.30GHz 4 core×2 processors, 96GB RAM, 180GB solid-state drive (SSD)×4 (RAID1+0) and 10Gbps network interface card (NIC). All servers in the second server set denoted as *Cloud-B* have Intel Xeon X5675 3.07GHz 6 core×2, 96GB RAM, 72GB hard-disk drive (HDD)×2 (RAID1) and 10Gbps NIC. We installed CentOS 7.2.1511 on all servers. The main difference between these sets is the disk configuration. All servers in Cloud-A have four SSDs, whereas those in Cloud-B have two HDDs. Then, Cloud-A has higher performnace than Cloud-B. These local disks called *ephemeral disks* are also used for VMs on the compute nodes. These different types of clouds lead to different performance in OpenStack, and they require different degrees of appropriate concurrent connection limits.

We used 22 servers in total for each cloud. We built three redundant OpenStack controller nodes with active–active manner by using one HAProxy node. Three of the servers were used for MariaDB and RabbitMQ cluster. We built the MariaDB cluster with active–standby manner by using one HAProxy node and RabbitMQ with active–active manner, which were running with data mirroring. Each cloud had 12 OpenStack compute nodes, one Network node and one NFS node. Notably, the hardware and software configuration of OpenStack was completely the same between Cloud-A and Cloud-B. We installed OpenStack Mitaka release on these 22×2 servers and used KVM as their hypervisor. The versions of the installed software are: Nova 13.1.0, Neutron 8.1.2, Keystone 9.0.2, Glance 12.0.0, HAProxy 1.5.14, MariaDB 10.1.12 and RabbitMQ 3.6.2.

2) Workload generation: To generate a synthetic workload in OpenStack, we used Rally [22], a benchmarking software for OpenStack services. Rally can generate workload with pre-defined useful scenarios. We selected one of the most basic scenarios that creates, boots and deletes VMs. This scenario has two main parameters: the number of concurrency (denoted as c) and the number of times to boot and delete VMs (denoted as t). This scenario first issues c *boot server* API requests followed by *delete server* API requests to Nova service, and a new *boot server* request is issued once one of c *delete server* API requests are completed. These operations are repeated until t VMs are booted and deleted. These two parameters c and t determine the degree and duration of the workload in OpenStack, respectively.

We fixed the synthetic workload for each cloud environment. We set $c = 200, t = 5000$ for Cloud-A and $c = 100, t = 2500$ for Cloud-B. We determined these values from preliminary experiments. Such values lead to performance degradation and can eventually complete all API requests. All booted VMs in our experiments have 1 virtual CPU, 2GB RAM and 20GB disk. The workload generated in this paper is stable. Although the workload in real IaaS clouds can fluctuate at times and several different scenarios are mixed, investigating with this stable workload is helpful to grasp the basic behavior and influence of admission control in OpenStack. Analysis based on real workload may be conducted in the future.

C. Effect of Admission Control in Cloud-A

We selected the statistics of the 95th percentile to evaluate the performance. The percentile-based evaluation can capture user-experienced performance, which is discussed in some existing studies [18], [25]. Fig. 3 plots the 95th percentile of the API response times of backend OpenStack controller nodes, the waiting times in the queue on HAProxy and the total response times for clients in Cloud-A. The sum of concurrent connection limit in Nova service (denoted as L) is changing in Fig. 3. We set the individual limits for the controller nodes to be the same as much as possible. Fig. 3 also plots error bars (vertical lines) that indicate the 95% confidence interval from 10 trials for each limit. We obtained these results from the *http logs* output by HAProxy. As shown in Fig. 3, the backend response times increase as L becomes large because the large number of concurrent connections causes resource contentions among the API requests. Meanwhile, the waiting times in the queue decrease as L becomes large because API requests have more chances to be admitted. The total response time is almost the same as their sum, and it takes a minimum value when L is around 100. However, the influence of L becomes small when L is larger than 40.

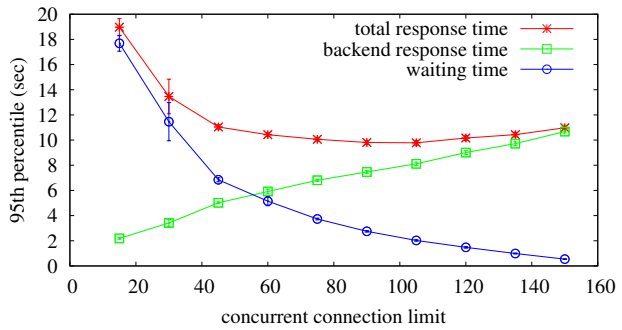


Fig. 3. Effect of the concurrent connection limit on API response times in Cloud-A obtained from HAProxy

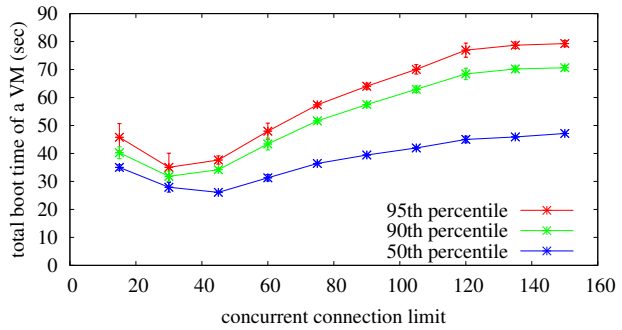


Fig. 4. Effect of the concurrent connection limit on the time to boot a VM in Cloud-A obtained from Rally

We then introduce the results obtained from Rally. Fig. 4 plots the statistics of the total time to boot a VM, when L is changing in Cloud-A. Fig. 4 indicates that the time to boot a VM is dependent on L. We can confirm from this figure that admission control in OpenStack can improve performance of OpenStack by more than 50% in an overloaded situation. When L is larger than 40, the effect is greatly large, which is completely different from that in Fig. 3.

The root cause for the difference between Figs. 3 and 4 is that some API requests in OpenStack result in asynchronous operations, which is the operations that is executed after responding to API requests. The *boot server* API requests are responded when the commitment of the VM creation is completed on the database as was explained in Subsection II-B. The remaining operations such as booting the VM are not included in the response times of API requests. This asynchronous scheme brings about the followings observations.

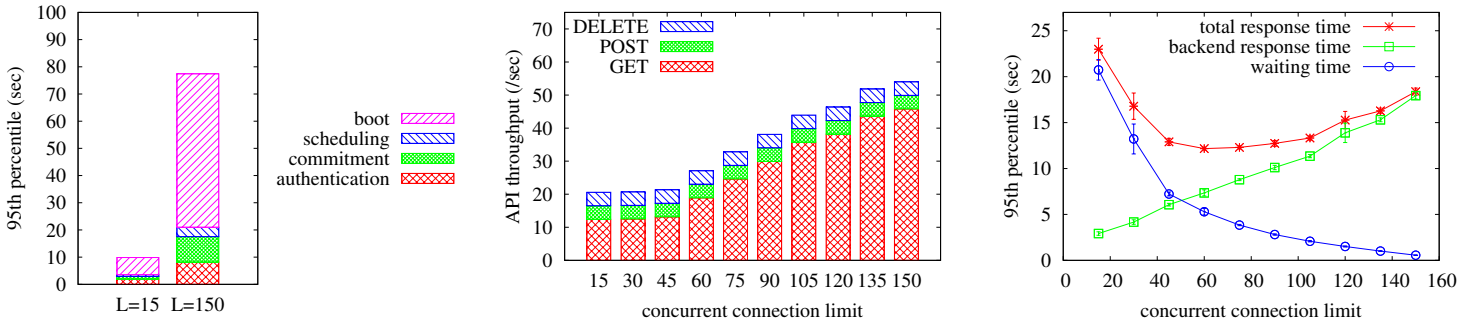
1) The durations of asynchronous operations account for a large portion of total boot times as the concurrent connection limit increases. This is not measured by the statistics of API response times.
2) OpenStack clients are required to execute periodical polling operations to check if the VM becomes active, which issues a large number of HTTP GET requests in OpenStack, especially when the durations of asynchronous operations are long. A long asynchronous operation causes workload distribution changes.

We can confirm the first observation from Fig. 5a, in which we breakdown the individual total boot time into four categories. We break them down by analyzing the timestamps of *debug* log messages output in the OpenStack controller nodes. All log messages involved by API requests have IDs called *request IDs* which are unique for each API request. We linked the associated log messages for each *boot server* API request based on the request IDs and analyzed their timestamps. The *authentication* in Fig. 5a is the time to authenticate the API request using Keystone and the *commitment* is the time until the writing of a VM entry in Nova database is finished. The *scheduling* is the time until an appropriate compute node is selected, and the *boot* is the remaining time spent in the compute node. We calculated the 95th percentile value for each category and stacked them in Fig. 5a. As shown in Fig. 5a, when L becomes large, a large portion of time is *boot*, which is the asynchronous operation.

Figs. 5b and 5c confirm the second observation. Fig. 5b shows the overall throughput of API requests for each HTTP method obtained from a benchmarking. The number of HTTP GET requests increases when L is large. The GET requests have much smaller response times than the POST requests. When the number of GET requests increases, the workload distribution characterized by statistics, such as the 95th percentile, is influenced. To eliminate the effect of workload distribution changes, we plot the API response times distinguishing the POST requests from all requests in Fig. 5c. The waiting time in Fig. 5c is almost similar to that in Fig. 3 because all requests for Nova service share the same queue in HAProxy. By contrast, the response times in Fig. 5c exhibit the different rate of increase from that in Fig. 3. These results show that the response times of POST requests, which more directly affects on the time to boot VMs, increase as L increases. This relationship is invisible when we monitor statistics such as the 95th percentile because of the large number of GET requests.

D. Effect of Admission Control in Cloud-B

We next show the results obtained from Cloud-B, which has lower performance than Cloud-A because of low-performance disk specification. Figs. 6–8 show the results we obtained from Cloud-B. The 95th percentile of API response times shown in Fig. 6 is getting better as L increases. This result is caused by the workload distribution changes because of the large number of GET requests. As shown in Fig. 6, the influence of GET requests in Cloud-B are larger than that in Cloud-A. This is affected by a long duration of asynchronous operation, which is confirmed from Figs. 7a. As shown in Fig. 7a, the *boot* time contributes greater portion of the total boot time compared with that in Fig. 5a due to the low-performance disk in Cloud-B. The *commitment* time in Fig. 7a is also affected by the low-performance disk and suffers some delay compared with that in Fig. 5a. We can confirm from Fig. 7b that the number of GET requests accounts for a large portion of all API requests in Cloud-B, which causes the behavior in Fig. 6. Fig. 7c plots the statistics of API response times distinguishing the POST requests, which exhibits a completely different behavior from that in Fig. 6 because of the large number of GET requests.



(a) Breakdown of the boot time (b) API request throughput for each HTTP method (c) Statistics of POST requests

Fig. 5. Detailed effects of the concurrent connection limit in Cloud-A

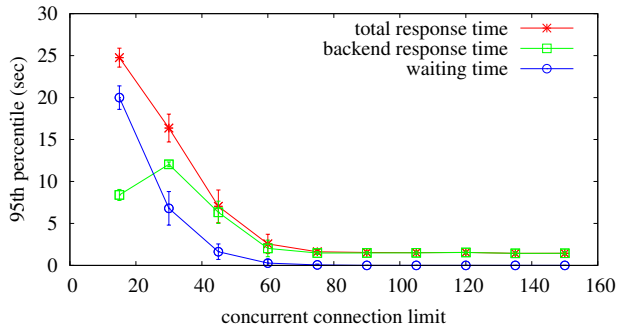


Fig. 6. Effect of the concurrent connection limit on API response times in Cloud-B obtained from HAProxy

The POST API response times in Fig. 7c exhibit the same tendency with the total boot times in Fig. 8. As shown in Fig. 8, in Cloud-B, L should be low (around 20) to avoid performance degradation. The performance degradation is saturated when L is larger than 40. This is because the low disk performance also leads to a large delay in the asynchronous operations to boot VMs. The workload scenario we used does not generate the newly *boot server* requests unless the asynchronous operations of early arriving requests are completed, which leads to a saturation of workload.

IV. Adaptive Admission Control

We propose an adaptive admission control algorithm for the concurrent connection limit in OpenStack. This algorithm is motivated by the results obtained from Section III: 1) the optimal concurrent connection limit is different depending on the cloud environment, which means that the concurrent connection limit should be adaptively determined based on the cloud environments, and 2) simple statistics such as the 95th percentile of API response times are not ideal to measure the current performance of the OpenStack cloud. The workload distribution changes depending on the concurrent connection limit, which means that simple statistics is unsuitable for guiding the appropriate concurrent connection limit.

A. Algorithm Design

We present an adaptive tuning algorithm for the concurrent connection limit, which is described in Algorithm 1, to adaptively determine the appropriate concurrent connection limit. The principle of our idea is to balance the waiting times in the queue and the backend response times. This idea comes from the trade-off relationship between these statistics as shown in Fig. 3. Severe performance degradation is more likely to occur when the balance of the waiting times and backend response times is broken. We define the *error* of the balance as:

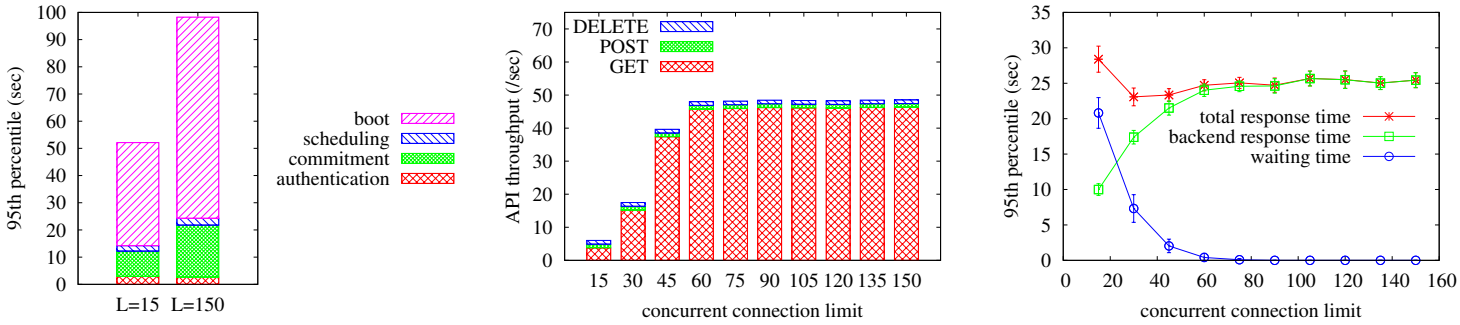
$$\text{error} = \frac{S_w - S_r}{S_w + S_r} \tag{1}$$

where S_w and S_r indicate the statistics of the waiting time in the queue on HAProxy and the backend response times, respectively. Our algorithm adaptively tunes the concurrent connection limit based on this error. This idea also presents a benefit in that it does not require a reference value of API response times. Therefore, our algorithm does not depend on the absolute value of the API response times. It can adapt to the cases shown in Fig. 6, in which the workload distribution is changing depending on the concurrent connection limit.

We adapted the proportional control technique from classical control theory with a specified proportional gain K_p, which determines how much the concurrent connection limit should be increased or decreased. Algorithm 1 first obtains the waiting time denoted as t_w and the backend response time denoted as t_r from each completed API request. When the cumulative number of completed API requests exceeds the pre-defined value (denoted as N), the statistics of t_w and t_r (denoted as S_w and S_r, respectively) are calculated. Our algorithm then determines the error from Eq. (1) and updates the limit L as $L \Leftarrow L + \delta L$ when error ≥ 0 and $L \Leftarrow \max(L - \delta L, L_{\min})$ when error < 0, where

$$\delta L = \lfloor |\text{error}| \times K_p \rfloor - (\lfloor |\text{error}| \times K_p \rfloor \mod u) \tag{2}$$

We set $L_{\min}$, the minimum concurrent connection limit, to prevent the limit from being set as 0 and u, the minimum unit of increased or decreased limit, to prevent trivial tuning of the limit. After the algorithm updates the limit, it sleeps for $\min(S_w, S_r) \times 2$ time to wait for the effect of tuning on the statistics of the completed API requests. After it sleeps, tuning is repeated in the same manner. When L is updated, individual limits (l_n) for the controller nodes are updated so as to be the same for all l_n as much as possible.



(a) Breakdown of the boot time (b) API request throughput for each HTTP method (c) Statistics of POST requests

Fig. 7. Detailed effects of the concurrent connection limit in Cloud-B

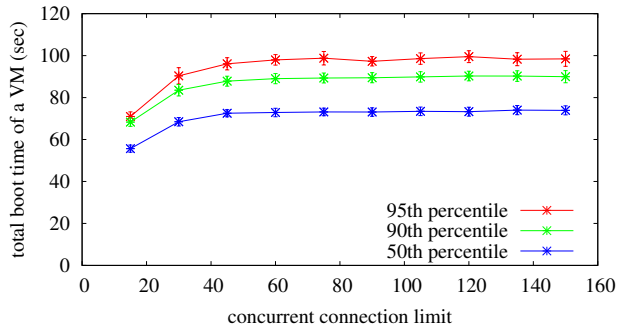


Fig. 8. Effect of the concurrent connection limit on total boot time of VMs in Cloud-B obtained from Rally

Algorithm 1 Adaptive tuning of concurrent connection limit

```
Minimum concurrent connection limit: L_min
Current concurrent connection limit: L ⇐ L_min
# of API requests that is calculated the statistics: N
Counter for new API requests: count
Set of backend response times: T_r
Set of queueing delay: T_w
Proportional gain for the adaptive tuning: K_p
while true do
    T_r ⇐ φ, T_w ⇐ φ, count ⇐ 0
    while count < N do
        if a newly completed API request exists then
            add t_r in T_r and add t_w in T_w
            count++
        else
            sleep 1 second
        end if
    end while
    S_r ⇐ calcstats(T_r), S_w ⇐ calcstats(T_w)
    error ⇐ (S_w − S_r)/(S_w + S_r)
    update L based on error, K_p, and L_min
    sleep (min(S_r, S_w) × 2) seconds
end while
```

B. Evaluation

We built a prototype of the adaptive limit controller which implements our algorithm (Algorithm 1) that runs on the HAProxy server for OpenStack controller nodes. The controller tails the *http logs* output by HAProxy and obtains the statistics of each completed API request. The controller adaptively updates the concurrent connection limit using the UNIX socket commands [15]. We set the parameter of our algorithm as $L_{\min} = 15, K_p = L_{\min}, u = 5, N = 100$ and calculate the 95th percentile of t_w and t_r.

Figs. 9 and 10 show the behavior of our adaptive limit controller in Cloud-A and Cloud-B. The upper graphs in these figures indicate the number of the concurrent connections and the requests in the queue, which are obtained from the *show stat* socket command of HAProxy. The middle graphs in these figures indicate the error calculated from Eq. 1, which is calculated every 10 s (a time bin). The bottom graphs are the 95th percentile of the waiting times, backend response times, and total response times for each time bin.

As shown in Fig. 9, our algorithm can automatically and adaptively tune the limit by balancing the waiting times and backend response times. The limit in Cloud-A converges at the near-optimal value of around 60. In addition, our algorithm can adapt to the Cloud-B environment, which requires a small limit on the concurrent connections. Fig. 10 indicates that the limit converges at around 20 in Cloud-B. The middle and bottom graphs in Fig. 10 have more fluctuating behavior than that in Fig. 9, because the arriving rate of API requests is small and a low number of requests exists in each time bin.

We next evaluated the overall performance of our adaptive limit controller as shown in Figs. 11a and 11b, in which we picked up some representative results from the cases that L is static ($L = 45, 150$ for Cloud-A and $L = 15, 150$ for Cloud-B) to compare the performance. These figures confirmed that our algorithm can avoid severe performance degradation in total boot time of a VM, especially in Cloud-A. Although the achieved performance is not optimal, severe performance degradation is avoided.

Finally, we evaluated the impact of the proportional gain K_p in Cloud-A. To show the effect of K_p more clearly, we selected 300 s from the beginning that the benchmarking started, and this is the average value of 30 time bins (each time bin has 10 s). The selected period is the phase that the controller is

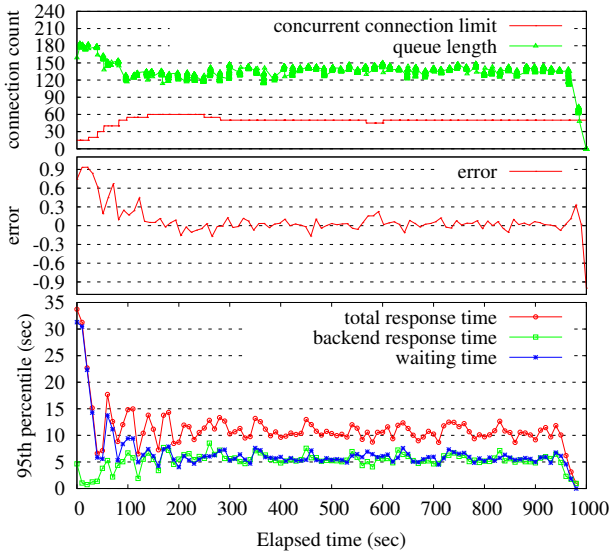


Fig. 9. Behavior of our adaptive limit controller and its effect in Cloud-A

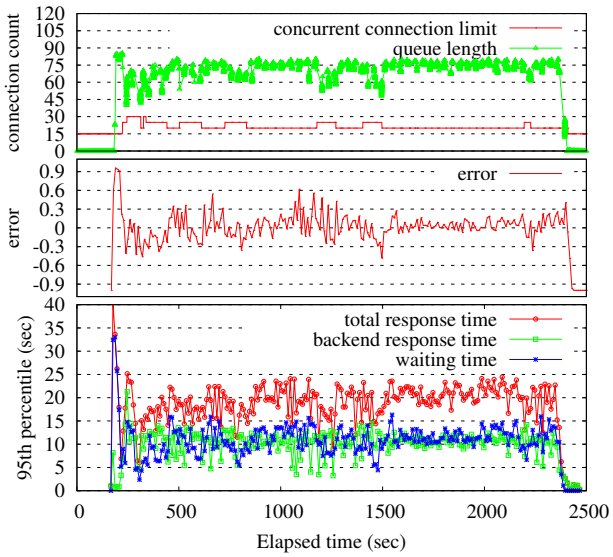


Fig. 10. Behavior of our adaptive limit controller and its effect in Cloud-B

actively tuning L and L is not converged yet as shown in Fig. 9. The error bars in Fig. 12 are the 95% confidence interval calculated from 10 trials. As shown in Fig. 12, the average absolute error calculated from Eq. 1 varies depending on K_p. However, its effect on the total API response time and total boot time is relatively small. As shown in Fig. 3, the place that can balance the waiting time and backend response time is less influenced by L settings. Although we should avoid setting a large K_p, this parameter is not remarkably sensitive for the performance of OpenStack.

V. Discussion

We investigated the influence of admission control in OpenStack API requests. Based on the results, we proposed an adaptive limit control algorithm, which achieved good performance on two different cloud environments. This section presents certain considerations that must be met to apply our adaptive limit controller to real production OpenStack clouds.

Admission control in this study uses a queue in the proxy server to delay the arriving requests when the system is overloaded. However, the waiting times in the queue and backend response times may become indefinitely long. When a drastically overloaded or faulty situation occurs, some arriving requests have to be dropped and the early overloaded message is sent as a response. This step, a well-known approach in production Web services, has been the focus of several existing studies [17], [19], [21]. To drop some requests, we have to set a timeout for the waiting time in the queue.

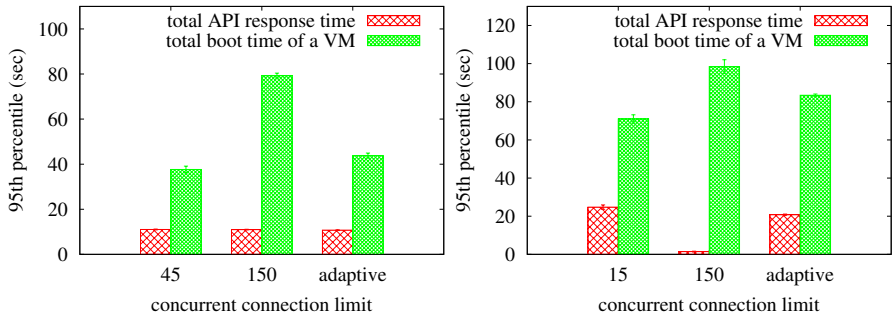
Asynchronous operations, such as the booting VMs, result in much workload in OpenStack. However, admission control based on the API response times does not directly address such operations. To sufficiently address these operations, we have to obtain the time to complete the asynchronous operations and tune the limit based on that. An approach based on the performance metrics such as CPU utilization on the compute nodes also may work good for such operations. However, the collection of these data takes much cost because a large number of the controller and compute nodes are running on production environments.

In some OpenStack clouds, setting a static limit works well based on preliminary experiments. Our adaptive limit controller has no guarantee that the best performance can be achieved, but it can help avoid severe performance degradation. The applicability of our principle idea in Algorithm 1 that balances the waiting times and backend response times must be more extensively investigated. However, the trade-off relationship between the two performance indicators are remained in all environments, and when a severe performance degradation occurs, there should be a large gap between these two indicators. The largest benefit of our adaptive controller is its adaptability. As shown in Section IV-B, our controller can achieve good performance in both OpenStack clouds, which have completely different abilities to serve the OpenStack service.

VI. Related Work

Based on the fact that many industrial developers have attracted much attention on OpenStack clouds, numerous recent studies have targeted the OpenStack clouds. Some of these studies focused on the performance of VMs running on OpenStack [10], [12] [26], [27]. Sharf et al. [10] studied network-aware VM scheduling in OpenStack. Karacali et al. [12] and Callegati et al. [26] measured and evaluated network performance with various of OpenStack network configurations. Vilaplana et al. [27] proposed an SLA-aware load balancing in Web-based application on cloud environments, and this strategy was implemented on OpenStack.

The existing work focusing on the services provided by OpenStack [9], [11], [13], [14] is more related to our present study. Sharma et al. [9] presented HANSEL, a root cause diagnostic system for faults in OpenStack. Fault diagnosis in HANSEL is conducted by monitoring control messages such



(a) Cloud-A (b) Cloud-B

Fig. 11. Effect of our adaptive limit control

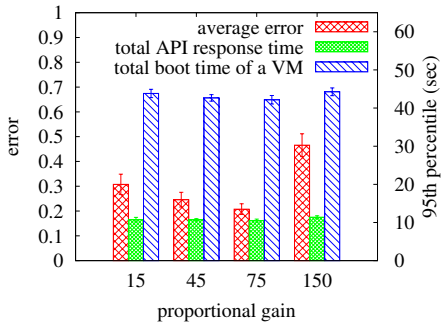


Fig. 12. Effects of the proportional gain in Cloud-A.

RPC calls and API requests in OpenStack. However, this work focused on the faults caused by human operations rather than its performance. Almasi et al. [13] evaluated the duration of VM creation compared with the different OpenStack releases by using CloudBench [28]. Paradowski et al. [14] evaluated the performance of OpenStack services compared with that of CloudStack [29]. OpenStack Rally [22] is a tool that is often used for performance evaluation of OpenStack service. Rally supports many kinds of benchmarking scenarios, including VM boot and delete, authentication, network creation, and volume attachment. These studies are limited to the evaluation of OpenStack service performance.

Although QoS management in OpenStack lacks detailed investigation, that in Web services has a long history supported many existing work. In the present paper, we introduce several studies focused on the admission control of Web services. Elnikety et al. [20] presented an admission control method. They estimated the capacity of Web sites by measuring the highest load that achieves maximum throughput of HTTP requests, and a newly arriving request is deferred in an admission queue when the admitting request will exceed the capacity. However, the capacity estimation is conducted offline, which leads to less adaptability of system configuration changes. Kamra et al. [17] and Liu et al. [21] investigated admission control for overloaded Web sites. Their admission control uses a technique from control theory with a reference response time combined with the M/G/1–PS queuing model to determine the drop probability of HTTP requests. Therefore, these studies require a reference value for response times. Ashraf et al.'s approach [19] utilizes extensively measured information, which includes the number of rejected, deferred, and aborted sessions and the number of overloaded servers to determine whether every newly arriving user session will be admitted. Their approach was evaluated with a discrete-event simulation. Welsh et al. [18] presented an overload management technique with the staged event-driven architecture model. They adaptively controlled the token bucket rate for admission at each stage based on the error between the 90th percentile of response time and a reference value.

Performance of admission control in Web services is also analyzed using the queueing model with a limited number of concurrent connections [30], [31], [32]. Cao et al. [30] modeled the performance of Web server using the M/G/1/K-PS model, in which the limited number of HTTP requests can be processed and the service discipline is processor sharing. Khazaei et al. [31] models Web server with the M/G/m/m+r model, which has a request buffer with size of r. By using this model, they analyzed the important performance indicators such as drop probability. However, these abovementioned studies require the estimation the parameters such as service demands of HTTP requests. Some studies have discussed parameter estimation [33], [34], but such a step results in complexity and computational cost that leads to decreased adaptability.

VII. Conclusion & Future Work

In this study, we focused on the admission control of API requests in OpenStack. We first investigated the basic effect of admission control in OpenStack by setting the concurrent connection limit. Our results confirmed that admission control can improve the performance of OpenStack services by more than 50% in an overloaded situation.

We also proposed a heuristic algorithm that adaptively controls the concurrent connection limit by monitoring the API response times from the backend OpenStack controller nodes and the waiting times in the queue in HAProxy. From our experimental evaluations, we found that our controller helped avoid severe performance degradation in OpenStack.

However, some asynchronous operations in OpenStack may cause difficulty in controlling the overloaded situation in OpenStack services. In addition, the synthetic workload generated in this paper is relatively stable and more realistic workload should be applied to our controller to confirm its efficiency. Lack of trace studies of real IaaS workload makes it difficult to conduct experiments with more realistic workload. Existing trace studies focus only on the workload generated by applications running on IaaS [35] and these help the studies about VM consolidation and placement in IaaS. In the future work, the workload traces of OpenStack services have to be investigated and conduct experiments, which are not stable and contain the workload of not only Nova but also the other OpenStack services.

REFERENCES

[1] (2016, November) Amazon elastic compute cloud api reference. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/APIReference/ec2-api.pdf

[2] (2016, November) Openstack api documentation. [Online]. Available: http://developer.openstack.org/api-guide/quick-start/index.html

[3] (2016, November) Openstack heat. [Online]. Available: https://wiki.openstack.org/wiki/Heat

[4] (2016, November) Ansible. [Online]. Available: https://www.ansible.com/

[5] (2016, November) Cloud foundry. [Online]. Available: https://www.cloudfoundry.org/

[6] (2016, November) Optimizing openstack for large scale cloud foundry deployments. [Online]. Available: https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/optimizing-openstack-for-large-scale-cloud-foundry-deployments

[7] (2016, November) Openstack. [Online]. Available: https://www.openstack.org/

[8] D. Freet, R. Agrawal, J. J. Walker, and Y. Badr, "Open source cloud management platforms and hypervisor technologies: A review and comparison," in *Proceedings of IEEE SoutheastCon 2016*, March 2016, pp. 1–8.

[9] D. Sharma, R. Poddar, K. Mahajan, M. Dhawan, and V. Mann, "Hansel: Diagnosing faults in openstack," in *Proceedings of the 11th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2015.

[10] M. Scharf, M. Stein, T. Voith, and V. Hilt, "Network-aware instance scheduling in openstack," in *Proceedings of the 24th International Conference on Computer Communication and Networks (ICCCN)*, August 2015, pp. 1–6.

[11] Y. Xiang, H. Li, S. Wang, C. P. Chen, and W. Xu, "Debugging openstack problems using a state graph approach," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2016, pp. 13:1–13:8.

[12] B. Karacali and J. M. Tracey, "Experiences evaluating openstack network data plane performance and scalability," in *Proceedings of the 15th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2016, pp. 901–906.

[13] G. Almsi, J. G. Castaos, H. Franke, and M. A. L. Silva, "Toward building highly available and scalable openstack clouds," vol. 60, no. 2-3, pp. 5:1–5:10, March 2016.

[14] A. Paradowski, L. Liu, and B. Yuan, "Benchmarking the performance of openstack and cloudstack," in *Proceedings of the 17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, June 2014, pp. 405–412.

[15] (2016, November) Haproxy documentation. [Online]. Available: http://cbonte.github.io/haproxy-dconv/

[16] (2016, November) Introduction to openstack high availability. [Online]. Available: http://docs.openstack.org/ha-guide/intro-ha.html

[17] A. Kamra, V. Misra, and E. M. Nahum, "Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites," in *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQOS)*, June 2004, pp. 47–56.

[18] M. Welsh and D. Culler, "Adaptive overload control for busy internet servers," in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003, pp. 43–57.

[19] A. Ashraf, B. Byholm, and I. Porres, "A session-based adaptive admission control approach for virtualized application servers," in *Proceedings of the 5th Internation Conference on Utility and Cloud Computing (UCC)*, November 2012, pp. 65–72.

[20] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites," in *Proceedings of the 13th International Conference on World Wide Web (WWW)*, 2004, pp. 276–286.

[21] X. Liu, J. Heo, L. Sha, and X. Zhu, "Adaptive control of multi-tiered web applications using queueing predictor," in *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2006, pp. 106–114.

[22] (2016, November) Openstack rally. [Online]. Available: https://wiki.openstack.org/wiki/Rally

[23] (2016, November) Mariadb galera cluster. [Online]. Available: https://mariadb.com/kb/en/mariadb/galera-cluster/

[24] (2016, November) Rabbitmq. [Online]. Available: https://www.rabbitmq.com/

[25] P. Lama and X. Zhou, "Autonomic provisioning with self-adaptive neural fuzzy control for percentile-based delay guarantee," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 2, pp. 9:1–9:31, July 2013.

[26] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea, "Performance of network virtualization in cloud computing infrastructures: The openstack case," in *Proceedings of the 3rd International Conference on Cloud Networking (CloudNet)*, October 2014, pp. 132–137.

[27] J. Vilaplana, F. Solsona, J. Mateo, and I. Teixido, "Sla-aware load balancing in a web-based cloud system over openstack," in *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*. Springer, December 2013, pp. 281–293.

[28] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. d. Silva, "Cloudbench: Experiment automation for cloud environments," in *Proceedings of 2013 IEEE International Conference on Cloud Engineering (IC2E)*, March 2013, pp. 302–311.

[29] (2016, November) Apache cloudstack. [Online]. Available: https://cloudstack.apache.org/

[30] J. Cao, M. Andersson, C. Nyberg, and M. Kihl, "Web server performance modeling using an m/g/1/k*ps queue," in *Proceedings of the 10th International Conference on Telecommunications (ICT)*, vol. 2, February 2003, pp. 1501–1506.

[31] H. Khazaei, J. Misic, and V. B. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 5, pp. 936–943, May 2012.

[32] V. Gupta and M. Harchol-Balter, "Self-adaptive admission control policies for resource-sharing systems," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009, pp. 311–322.

[33] J. F. Prez, G. Casale, and S. Pacheco-Sanchez, "Estimating computational requirements in multi-threaded applications," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 264–278, March 2015.

[34] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, "Estimating service resource consumption from response time measurements," in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, October 2009, pp. 48:1–48:10.

[35] A. Beloglazov and R. Buyya, "Openstack neat: a framework for dynamic and evergy-efficient consolidation of virtual machines in openstack clouds," *Concurrency and Computation Practice & Experience*, vol. 27, no. 5, pp. 1310–1333, April 2015.