A Solution to Resource Underutilization for Web Services Hosted in the Cloud

Dmytro Dyachuk and Ralph Deters

MADMUC lab
Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan
S7N 5C9 Canada
dmytro.dyachuk@usask.ca, deters@cs.usask.ca

Abstract. At the moment the service market is experiencing a continuous growth, as services allow easy and quick enhancing of new and existing applications. However, hosting services according to a common on-premise model is not sufficient for dealing with erratic, spike-prone service loads. A new more promising approach is hosting services in the cloud (utility computing), which enables dynamic resource allocation. The last provides an opportunity to meet average response time requirements even in case of long-term fluctuating loads. Unfortunately, in the presence of short term fluctuations the resources utilization has to stay under 50% in order to achieve response time of the same order as job sizes.

In this work we suggest to compensate the problem of underutilization caused by hosting low-latency services by means of allocating the remaining resources to time insensitive service requests. This solution uses load balancing combined with admission control and scheduling application server threads. The proposed approach is evaluated by means of experiments with the prototype conducted with Amazon's EC2. The experimental results show that the servers utilization can be increased without penalizing low-latency requests.

1 Introduction

After being introduced in 2000, Web Service technology [12] started penetrating the world of the distributed computing. Web Services excelled in compare to the other competitors mostly due to using platform independent XML based technologies. Simple Object Access Protocol (SOAP) [22] is employed as a message encoding protocol and Web Service Description Language WSDL [13] is used as a language for specifying the protocol of interacting with services. XML based protocols enable implementing large platform and language independent distributed systems. Moreover, Web Services as opposed to Common Object Request Broker Architecture (CORBA) middleware offer loser coupling and use simpler message encoding protocol, which is supported a larger number of programming languages. Even though Web Services may seem a superior approach to CORBA,

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 567–584, 2009.

the heterogeneity of Web Service brought by XML adversely impacts their performance. And thus CORBA is still often favored for implementing low-latency and/or real-time applications [14].

At the moment the service market is experiencing a continuous growth, as services allow easy and quick enhancing new and existing applications. Among the offered services, a substantial percentage is constituted by low-latency data services[1]. Some of these services have real-time properties implying that a response from a service will contain information which is valid for a predefined amount of time (i.e. a stock quote service). Thus, it is crucial for real-time data services to ensure that a request is being processed within a given time constraint and the response from a service will be delivered in a timely manner.

In this paper we address the issue of resource underutilization for low-latency data services hosted in the utility computing cloud. Low-latency service can be defined a service whose response time closely approaches request execution time, where request execution time is the amount of time for processing a single request.

Request arrival rate has a strong impact on services response time, the higher is the request rate the higher is the response time. In order to keep the average response time under a certain threshold the arrival rate has to be limited as well. Unfortunately, in order to achieve low-latency response time the utilization of a server hosting a service should stay markedly lesser than 50% [fig. 2, 3].

In this work we suggest eliminating the problem of underutilization caused by hosting low-latency services by means of allocating remaining resources to time insensitive service requests. Time insensitive requests can represent a service with no response time guarantees, for instance low-priced services, services used for free public consumption, offered without Service Level Agreements(SLA), etc. The solution consists of three layers: load balancing, admission control/request scheduling and thread scheduling. A load-balancing component distributes incoming requests among servers in a server pool. After that the admission control component decides if a service request can be accepted into the service, otherwise it will be put into a waiting queue, where time critical requests (low-latency) are preceding the time insensitive (regular) requests. As for the last step the priority of a thread assigned to handling an accepted request is being altered. In this work we present experimental results conducted with Amazon's EC2[1] and compare them with reference results obtained from the queuing theory.

The paper is structured as follows. In the second section we discuss three mainstream approaches for hosting services which are exposed for massive public consumption. The underutilization problem emerging while attempting to guarantee low-latency response times is presented in section three. An approach for tackling the problem of underutilization as well as its implementation details can be found in section four. Results of the experiments are located in section five. The last three sections contain a conclusion as well as an outlook on the related and future work.

[1] Companies like Strike Iron (http://www.strikeiron.com/) provide a number of services for accessing geographical, financial data, etc.

2 Hosting Services

At current, three main approaches can be distinguished for hosting services.

- On-premise (or in-house) hosting
- Utility computing
- Platform-as-a-Service hosting

On-premise hosting implies that a service provider is responsible for purchasing physical equipment, like servers, network bandwidth in order to host a service. This model has a set of advantages as a service provider has the flexibility of choosing the most suitable network topology and appropriate hardware configurations as well as an operating system. However, one of the key providers responsibilities is to perform an analysis of possible service loads and allocate an appropriate number of servers. The drawbacks of this methodology consists in the fact that often capacity planning is performed on the base of the worst case analysis. The last suggest allocating resources with regard to the maximum load. And as a result most of the time servers can be highly underutilized, what in its turn negatively reflects on the running costs. Moreover, on-premise infrastructure is vulnerable to erratic workloads which can be characterized by the presence of spikes in request arrival rates. The spike arrivals often yield to a significant response time increase or even denying services for some requests. Besides that, in case clients are distributed over a large space area on-promise computing requires significant investments for implementing low-latency services. The communication with services happens over the Internet and the time for delivering a request to services itself has a strong correlation with the percentage of lost packets, packet latency and network throughput itself. All the aforementioned factors depend on the physical distance from a service to a client. Thus, the larger the distance is the more packets are being lost, more hops have to be made and lesser is the throughput. Consequently, a service provider is behooved to place servers in different parts of a continent or even on different continents.

Utility computing in contrast to on-promise hosting provides the ability of "renting' servers in the cloud instead of purchasing physical machines. Thus, utility computing is often seen as an affordable alternative to the expensive on-premise approach. With utility computing it became possible to increase/decrease server pools dynamically with regard to service load (request arrival rate). Utility computing also looks more attractive for implementing low-latency services, as service provider can decide on the geographical location of server pools and host a service closer to clients. For instance, with Amazon's EC2 [1] the resources can be allocated in any of the four availability zones: three regions in North America and one in Europe. However, it is important to realize that service provider is fully responsible for implementing load balancing as well as fault-tolerance mechanisms.

Utility computing services may differ in terms of supported platforms as well as in term of the resources offered. But usually the choice of servers configuration is limited to a number of the predefined ones. The situation is similar in terms of the supported operating systems. Utility computing is viewed as a

commodity and thus the price for its uasage is based on the server hours, amount of incoming/outgoing traffic, volume of persistent storage, etc.

Platform-as-a-Service (PaaS) is the latest technology in comparison to the other two. PaaS providers usually take responsibility for maintaining underlying operating system, allocating sufficient amount of resources as well as implementing load balancing. However, there is a number of constraints imposed on hosted services. Those constraints emerge mostly due to security precautions, ensuring proper isolation when several services are sharing a host, as well as correct operating of services and enabling their management. For example, Google-App [4] engine allows hosting applications written in Python with no access to file I/O, the running time of each request is limited to one second and the proprietary storage (BigTable[4]) has to be used as a means for storing persistent data. The costs for hosting a service in such environment can be based on the number of requests served, consumed CPU time, amount of storage used, etc.

The last two methods for hosting services are often referred as hosting service in the Cloud. We believe that due to numerous limitation imposed by PaaS providers service providers will favor utility computing.

3 Resource Utilization in the Cloud

Works [25,26] have shown that loads on the Internet systems are highly irregular, low request arrival rates frequently intermittented by high request rates. The irregularity introduced by load adversely reflects on the response time, causing abnormally high service response time during the peak load and resource underutilization during the low loads. The abnormally high response time can negatively affect the reputation of a service or even cause waiting timeouts on a client side, what in its turn will decrease perceived reliability of the service.

Utility computing made it possible to adjust systems capacities dynamically to load fluctuations and thus to avoid the aforementioned problem. However, there is a latency associated with allocating new servers, and thus this approach allows compensating loads fluctuations only within 10-20 minutes range. Therefore, it would be a natural solution to allocate more resources, just enough to sustain the load, while the new servers are being added.

The number of request arrivals to an Internet system can be represented by means of a Poisson process[20]. A Poisson process sufficiently describes short-term arrival rate fluctuations and therefore we employ it for describing the number of request arrivals per time unit.

Let's define job size S or service time the amount of time required for processing a specific request(job) on an idle resource. For a CPU bound service this can represent the amount of the CPU time required for processing a request. Service response time T will be used as the time required for processing a request in the presence of other requests, response time T encompasses the sum of waiting time W and service time S: $T = W + S$.

Figure 1 contains a chart showing the analytical results for modeling the response time of a service running on a single core machine with an average

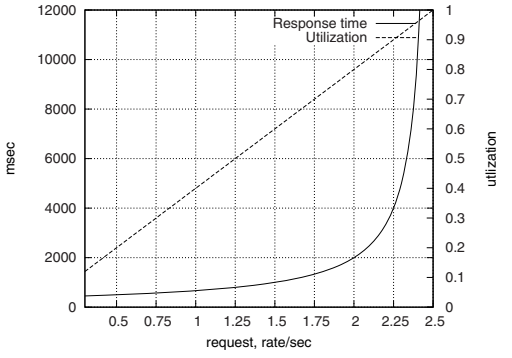

Fig. 1. Response Time (M/M/1), s=400msec

service time of 400 msec. The results are obtained according to the following formula $\overline{T} = S/(1 - \lambda S)$ [11], where λ is the average request arrival rate and $\overline{T}$ is the mean response time[2].

A chart on figure 2 shows the dependence of a service response time from its request arrival rate. As one can see on both charts [fig. 1 and 2] response time rapidly increases as the load approaches the capacity limit (2.5 req/sec). The utilization of the CPU by the service and analytical utilization can be seen on charts [fig. 1 and 3]. Therefore, for service providers it becomes of a great importance to keep the arrival rate so that utilization will not exceed 70%. Thus, on average 30% of the resource will have to idle. The situation gets worse if the service provider intends to provide low-latency services. In this case the arrival rate most likely will have to be limited to 20%-30% of the service's capacity. In this paper we propose allocating the remaining percent of the resource to the time insensitive requests and thus increase the utilization of service hosts without affecting the critical requests.

One can distinguish two different approaches for prioritizing critical requests, preemptive and non-preemptive [11]. In the preemptive case service suspends processing non-critical requests as soon as a critical request arrives. In the non-preemptive case the requests are placed in a waiting queue and the critical requests are being served first. The obvious advantage of the first approach is that the critical job does not have to wait for a service to become free.

An impact of the waiting time can be seen on figure 4. The chart shows how preemption affects a service with a mean service time of 400 msec. Critical requests are arriving at the average rate of 0.5 req/sec, the non-critical requests arrive at the rate increasing from 0 to 2.0 req/sec, the x axis shows the total arrival rate. As it can be seen in non-preemptive case critical requests response time increases with the growth of the regular requests arrival rate, while in the preemptive case the arrival rate of the regular requests does not affect the

[2] In spite the fact that this formula is derived for M/M/1/FCFS queuing system it is also valid for M/M/1/PS, which more accurately models processor sharing systems [11].

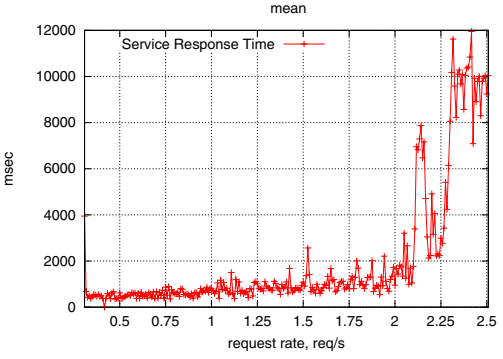

Fig. 2. Service Response Time

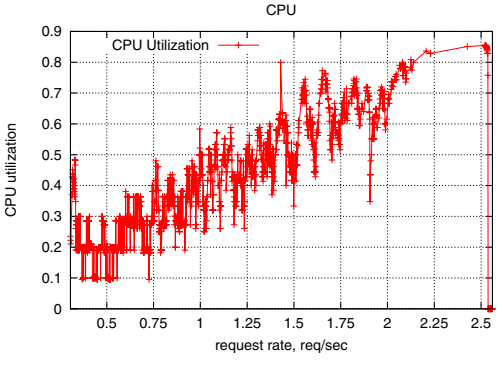

Fig. 3. CPU Utilization

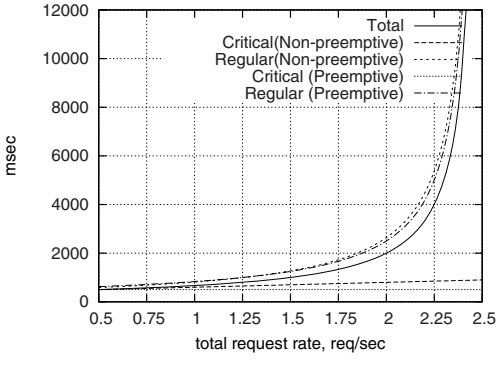

Fig. 4. Response time of preemptive and non-preemptive service

response time of the critical ones. Therefore, this work is based on preemptive priority scheduling.

4 Solution Architecture

In order to implement a scalable service hosted in the Cloud usually a two-tier architecture is employed. The first tier consists of a load-balancing reverse proxy. The second tier is constituted of a set of servers forming a server pool [fig. 5]. The key proxy roles is to hide servers from a direct client access and to distribute load among servers. The proxy is governed by a load balancing policy. The most common load balancing policies include but are not limited to Weighted Round Robin [17], Least Connections, Sticky Sessions, etc.

In this work we employ the Round Robin policy due its wide popularity and low overhead. Round Robin operates by equally distributing the load by forward incoming requests to servers in turns. Thus after a request has arrived at the load balancer it will be forwarded to a server.

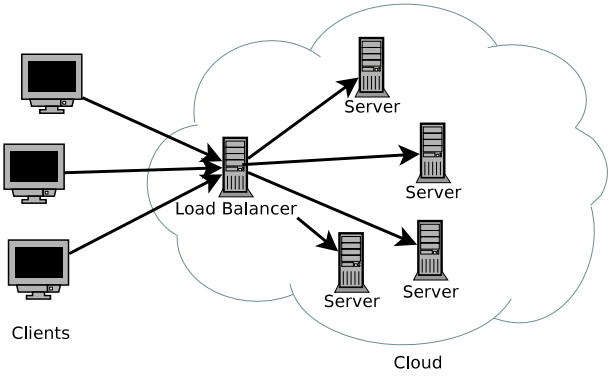

Fig. 5. Architecture for services running in the cloud

In case a request can not be accepted into service immediately it will be placed into a waiting queue. The acceptance decision is made by an admission control mechanism which is used as a means for preventing thrashing effect [15]. Thrashing can emerge in systems where requests are handled concurrently in threads [16]. Requests residing in the queue are ordered according to their priorities. After the request has been admitted in service, the priority of a handling thread is altered.

Each request has priority p and weight w associated with it. The last two parameters are extracted from the SOAP header [fig. 7] which is mandatory for every service request. The maximum priority is denoted as $\hat{p}$ and is used to mark the most critical requests, which do not have to wait for an admission.

Weight is a parameter which can be used for managing admission control on a fly, for instance for implementing adaptive admission control [16]. In this work

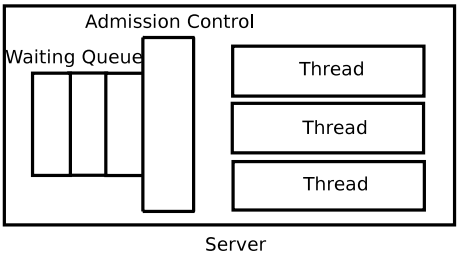

Fig. 6. Architecture of a hosting server

all the requests have equal weight w_c thus the maximum number of requests which can be processed in parallel is R_0/w_c, where is R_0 is a constant describing server's capacity. If to note R as the amount remaining resources then the algorithm for implementing scheduling and admission control is the following:

if *there is a request* i *in the waiting queue, such that* $p_i = \hat{p}$ ***then***
  $R = R - w_i$
  start processing request i *and set the priority of a processing thread to* p_i
else
  i = *request with the highest priority in the waiting queue*
  if $w_i \leq R$ ***then***
    $R = R - w_i$
    start processing request i *and change the priority of a processing thread to* p_i
  end if
end if

After request x has been processed the amount of the remaining resources is updated: $R = R + w_x$. The admission control and scheduling is invoked every time a new request arrives or a request leaves the service.

```
<soapenv:Header>
<tns:weight xmlns:tns="http://jobid.usask.ca"> 10 </tns:weight>
<tns:priority xmlns:tns="http://jobid.usask.ca"> 90 </tns:priority>
...
</soapenv:Header>
```

Fig. 7. SOAP header

5 Experiments

We have built a prototype in order to test if the proposed approach allows better utilization of service servers without impacting response time for low-latency requests.

5.1 Prototype Implementation

The prototype was implemented in Java and C. We used Apache Tomcat 5.5 [3] as a servlet container, Apache Axis2 as a Web Services middleware, HAProxy 1.3.12 [5] as a load balancer, and Linux (kernel 2.6.21.7-2.fc8xen) as an operating system. Sun's JRE 1.6 [6] was chosen as a Java runtime.

HAProxy is a high performance solution for implementing load balancing while supporting high-availability and proxying. HAproxy can operate on TCP as well as on HTTP levels. In this work we assume that services are stateless and therefore incoming service requests can be forwarded to any of the servers hosting a service and thus use regular HTTP load balancing.

Admission control and waiting queue scheduling mechanisms were implemented in Java and deployed as an Axis2 module, which can be used with an arbitrary Axis2 Web Service.

It is important to note that in modern Java runtimes user threads are mapped one-to-one to kernel threads. Consequently, user thread priorities are directly translated into the kernel thread priorities. Unfortunately, Sun's JRE Linux provides very limited abilities for modifying priorities of user threads within Java environment itself. It is not possible to set threads priorities to real-time nor alter the thread scheduling policy. Therefore, we had to develop our own Linux native library in C for setting arbitrary kernel thread priorities. The library interacts with scheduling and admission control module through Java Native Interface.

As it was mentioned before, after a job has been accepted it will be processed in a thread. During preliminary experiments we discovered that changing threads priorities at runtime results in additional delay and creates extra overhead. Therefore, the default configuration of Tomcat server was altered to contain two connectors and consequently two thread pools. One connector was responsible for serving regular requests and the other was accepting low-latency ones. The priorities in the thread pools were set immediately after the server has booted up and were not altered during the runtime.

One of the key goals in implementing the proposed idea was to decouple the described tools from a specific application server. At current the module was tested with Tomcat 5.5 which employs leader-follower model for handling incoming connections. But the developed scheduling module can be adapted for integrating with master-slave connectors without extensive modifications.

5.2 Prototype Evaluation

The proposed solution was tested with a Web Service running on Axis2+Tomcat platform. The services business logic consisted of recursive Fibonacci numbers calculations. Calculating Fibonacci numbers is a stack intensive operation which mimics method invocations typical for most services. The service contained a single method which required 400msec of CPU time. In order to eliminate the noise created by various jobs size and focus on the impact of arrivals and scheduling the jobs sized were set to constant size.

Table 1. Loads created by critical and regular requests

	One Server	Two Servers	Four Servers
$\lambda_{critical}$	0.5	1.0	2.0
$\lambda_{regular}$	0.5..2.0	0.5..4.0	0.5..8.0
λ_{total}	1.0..2.5	1.5..5	1.5..8.0

Table 2. Response time statistics (low-latency requests)

	One Server	Two Servers	Four Servers
Mean	443.4	405.2	402.5
95%	648.15	484.50	434.04
Max.	1105.1	627.0	594.6

Load balancer as well servers were running on Amazon's EC2 [1] single-core machines with 1.7 GB of RAM (m1.small). The service infrastructure encompassed a load balancer and a server pool. The load balancer was governed by the Round Robin scheduling policy. Three different cases analyzed when the service server pool consisted of one, two and four servers.

Requests were arriving according to a Poisson process. In each experiment the service was exposed to an increasing load created by regular requests [tbl. 1], while the low-latency requests were arriving the same rate. Moreover, low-latency requests were allocated 20% of the resources. while the rest was available for handling time insensitive requests. Such a load pattern was deliberately chosen to reflect the situation when the servers pool size is determined by the low-latency request rate. Thus, the load imposed by the regular request plays the second role and may fluctuate.

Each experiment consisted of nine phases corresponding to different arrival rates and lasted for one hour. The x axis on all charts depicts the arrival rates of regular requests. In each experiment the total load approached the services capacity and thus at the last phase the service was operating under 90% CPU utilization.

Tsung 1.3 [8] was used as a load generator. Each data sample on charts corresponds to an average value collected over a ten second time interval. Clients behavior followed an open model, which means that response time from a server did not affect arrivals of the next requests. CPU utilization was measured using standard *ps* tool available in any Linux distribution.

Single Server Case: As expected, the response time for regular requests started rapidly increasing as the total arrival rate began reaching the maximum request rate (2.0 req/sec for regular or 2.5 req/sec in total) [fig. 9]. However, the processing of the low-latency requests has not been disrupted and the service processed those requests on average in 443.4 msec [fig. 8, tbl. 2]. It is important to note, that according to the queuing theory in the preemptive case the higher priority jobs are not affected by lower priority ones. In reality, hoisting Linux threads priorities to real-time does not guarantee full preemption, unless a real-time Linux

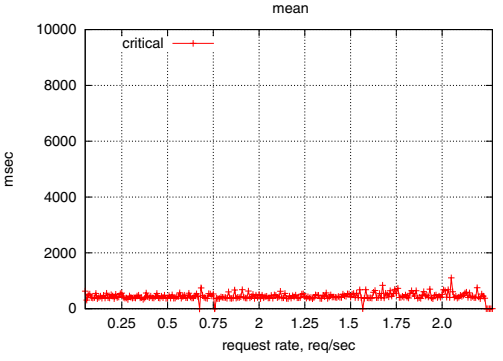

Fig. 8. Response time of critical (low-latency) service invocation. Single server case.

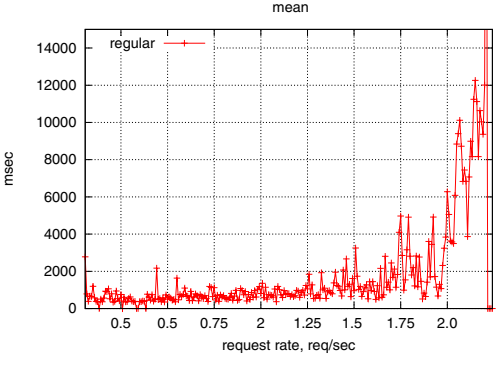

Fig. 9. Response time of regular service invocation. Single server case.

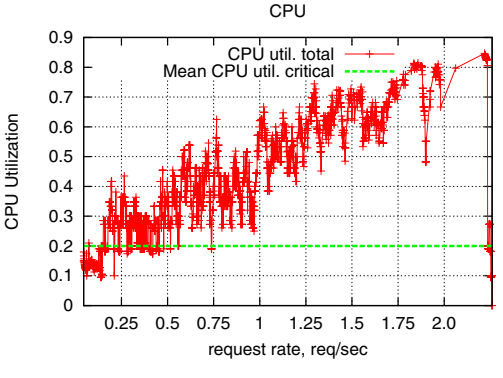

Fig. 10. CPU utlization. Single server case.

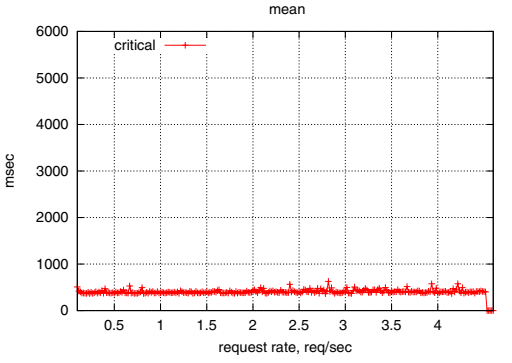

Fig. 11. Response time of critical (low-latency) service invocation. Two servers case.

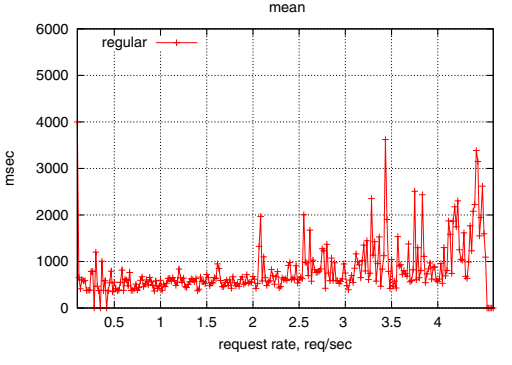

Fig. 12. Response time of regular service invocation. Two servers case.

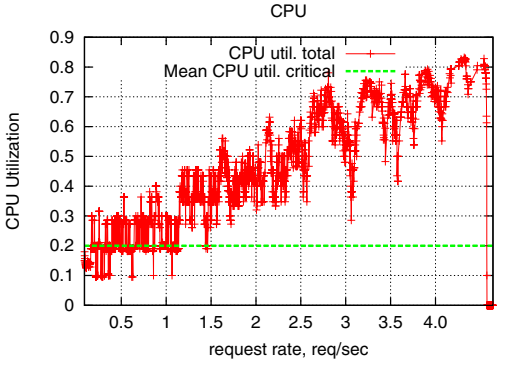

Fig. 13. CPU utilization. Two servers case.

kernel is used. Unfortunately, running Linux with a real-time kernel in the Amazon's EC2[1] at the moment is technically impossible, as the only allowed kernels are the ones provided by Amazon[1]. However, the results show that the mean

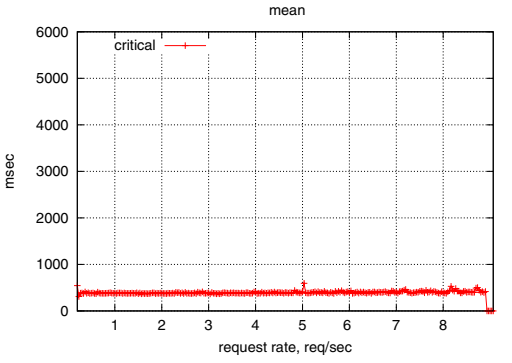

Fig. 14. Response time of critical (low-latency) service invocation. Four servers case.

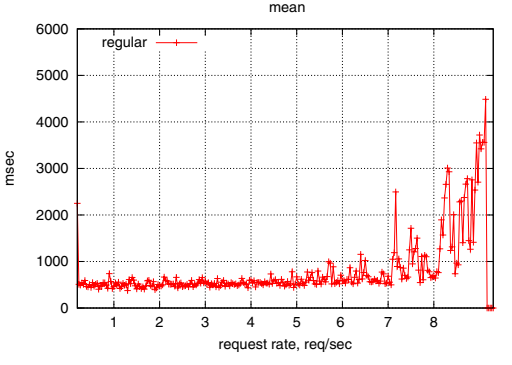

Fig. 15. Response time of regular service invocation. Four servers case.

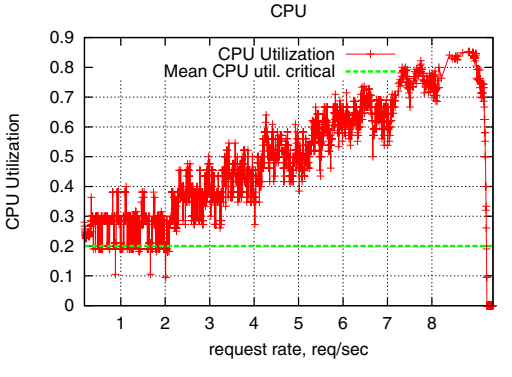

Fig. 16. CPU utlization. Four servers case.

response time for the the low-latency requests was 443 msec what is only 10.1% larger than its average size obtained using a queuing model [fig. 4] of 400.4 msec. Besides that 95% of critical requests were served in lesser than 648 msec [tbl. 2].

At same time response time of the regular requests increased from 400 msec to 14000 msec [fig. 9] or approximately 35 times.

As expected CPU utilization has increased from the initial 20% to 90% [fig. 10]. The X axis reflects the arrival rate of the non-critical requests. As it can be seen the fluctuations in the request arrivals resulted in the irregularities in the CPU utilization. Even when the total CPU utilization has reached 90%, when the time insensitive service invocations were occurring at the highest rate the response time for time sensitive service invocations remained unchanged [fig. 8]. As a result we can conclude that the suggested mechanism can be used for isolating the CPU bound critical load from the non-critical one.

Two Servers: In this case the server pool contained two servers. As it can be seen from the charts on figures 11 and 12 the service was processing all low-latency requests by 627 msec, moreover 95% of the request were served in lesser than 484.50 msec. It is important to note that with the increase of the server pool the difference between the mean response time in the reference result (400 msec) and actual result has shrunk. Since there were two servers load balancing helped to mitigate the fluctuations emerging in the Poisson arrivals.

The similar behavior is exhibited by regular requests. Their response time is still rapidly increasing as the load approaches 90% [fig. 12], nevertheless the growth rate is lesser than in the first case. The graph on figure 13 shows CPU utilization on one of the servers from the pool. The other servers experienced the same load and therefore the behavior of their CPU utilization is practically the same.

Four Servers: After scaling the load twice and doubling the resource pool the behavior exposed by critical and regular requests remains the same [fig. 12,13]. Besides that the 95% percentile of the response time for low-latency requests has got smaller as opposed to the single and two server cases. The CPU utilization has showed the same behavior as in the previous two cases [fig. 16].

6 Related Work

In work [27] Urgaonkar et al. presented an analytical model based on results from the queuing network. The model allows precise approximation of the mean response time from multi-tier systems and can be employed for performing dynamic capacity provisioning. The general principles which can be employed for performing service environment provisioning in order to meet response time guarantees were outlined by Ludwig in [21].

Another branch of works [24] focused on providing differentiated response time of a service running on a single server. In this work Siddaharta et al. implemented a proxy (Smartware) which intercepts the incoming requests, parses and classifies them according to user and client (device) type. In contrast to our work Smartware operates is a non-preemptive scheduler which is governed by a a randomized probabilistic scheduling policy derived from the lottery scheduling [28]. The experiment showed that the proposed approach outperforms the classical non-preemptive priority scheduling [11].

Real-time Java [7] provides tools for building real-time applications in Java. The environment provides a full spectrum of necessary tools, i.e. creating preemptive threads, priority inheritance protocol [23]. Unfortunately, besides the fact that this is a not a free product, it can be used only with Solaris or real-time Linux operating systems. The latter requires a special kernel, which is not supported by Amazon's EC2.

Levy et al. in [18] presented an approach for performance management of cluster-based Web Services. The authors addressed an problem of meeting response time guarantees for critical requests in cluster environments. The proposed solution employs non-preemptive Weighted Round Robin [17] load balancing schema in which weights are adjusted dynamically with respect to changes in the request arrivals rates.

Abdelhazer et al. in [10,9] proposed using control theory in order to meet predefined QoS requirements. The authors considered the problem of ensuring QoS in the environment with the fixed amount of resources. Content adaptation combined with admission control were used as a means for protecting servers from overloads. The conducted experiments showed that admission control which implements non-preemptive priority scheduling can be used as sufficient means for providing differentiated throughput.Moreover, this work is closely related to ours as it allows allocating underutilized resources to requests without performance guarantees. However, the authors focused on the throughput. It was demonstrated that their approach allows preserving the throughput of time-sensitive (premium) requests while allocating unused resources to processing time insensitive requests. The authors have enforced prioritization only at the admission control phase and thus implemented non-preemptive priority scheduling policy. Our approach differs by extending admission control with altering priorities of the threads handling incoming requests. Therefore, we implemented preemptive scheduling. In the preemptive case high priority requests have smaller waiting time as in case the resource is busy they do not have to wait. As a result, preemptive approach allows even better isolation of the high priority requests [sec. 2].

In [19] Lin and Dinda addressed the problem of underutilization by co-allocating virtual machines on the same physical host. The proposed idea is implemented as a user level scheduler VSched. VSched allows hosting virtual machines which are running applications with stringent time constraints together with virtual machines time insensitive batch processes. This implements preemptive scheduling with the isolation of time sensitive applications on the operating system level.

7 Future Work

In the future experiments we would like to replace the existing HAProxy load balancer with our own implemented in Erlang. HAProxy is a highly efficient and scalable load balancer. However, it was primarily designed for handling HTTP traffic. We plan to reimplement the basic functionality of HAProxy and to extend it in order to address specific needs of Web Services and cloud computing. Using Erlang would allow minimizing the overhead caused by load balancing and

scheduling, while having our own component would give freedom for realizing more complex load balancing/scheduling policies.

In this paper we have addressed the issue of underutilization emerging in CPU bound services. In the future we would like to overcome this limitation and to consider services which may have other bottlenecks affecting service's capacity, such as memory, disk I/O, etc.

In this work we have considered the case where critical requests were guaranteed to be allocated sufficient amounts of resources in order to maintain their mean response time at the desired level. We would like to augment this approach by considering cases when there could be multiple classes of service with different Quality of Service objectives.

8 Conclusion

Hosting services in the utility computing cloud allows dynamically scaling a server pool running a service. As a result, it is possible to allocate a sufficient amount of resources to accommodate oscillating load [20]. Unfortunately, adjusting the pool size allows compensating only long-term fluctuations while the short term ones can be compensated only by allocating additional resources. Consequently, service providers must keep service utilization under 50% to provide services whose response time approaches CPU time required for processing the request. Which means that 50% of the time resources are idling. In order to compensate this loss we suggest to assign the remaining resources to processing time insensitive requests. i.e. service offered without Service Level Agreements.

The suggested idea was implemented in the form of a two-tier scheduling system integrated with load-balancing. The proposed system transparently schedules processing order of the incoming requests, while avoiding thrashing using admission control mechanisms. Moreover, the arrived requests are handled in threads running at different priorities. By altering thread priorities and performing special admission control it became possible to achieve higher server utilization without affecting the response time for low-latency requests.

A prototype of the proposed approach was implemented as an Axis2[2] module and thus can be easily integrated with an arbitrary Axis2 Web Service. The prototype was evaluated by means of experiments conducted over a scalable CPU bound service hosted in Amazon's EC2[1]. The service experienced short term load fluctuations which were modeled using a Poisson process. The load created by low-latency was fixed at 20%, while the time insensitive requests were using from 10% to 70%. The experimental results showed the response time of low-latency requests was unaffected even when the total load exceeded 90%.

Acknowledgements

The authors would like to thank the Discuss lab at the University of Saskatchewan, especially George Biswas for using some of their software libraries.

References

1. Amazon Elastic Compute Cloud (Amazon EC2), http://www.amazon.com/gp/browse.html?node=201590011
2. Apache Axis, http://ws.apache.org/axis/
3. Apache Tomcat, http://tomcat.apache.org/
4. Google App Engine, http://code.google.com/appengine
5. HAProxy. The Reliable, High Performance TCP/HTTP Load Balancer, http://haproxy.1wt.eu/
6. Java SE Runtime Environment 6.0, http://www.java.com/
7. Sun Java Real-Time System, http://java.sun.com/javase/technologies/realtime/rts/
8. Tsung, http://tsung.erlang-projects.org/
9. Abdelzaher, T., Shin, K., Bhatti, N.: User-level qos-adaptive resource management in server end-systems (2003)
10. Abdelzaher, T.F., Shin, K.G., Bhatti, N.: Performance guarantees for web server end-systems: A control-theoretical approach. IEEE Trans. Parallel Distrib. Syst. 13(1), 80–96 (2002)
11. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications, 2nd edn., May 2006. Wiley/Blackwell (2006)
12. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web services architecture. Technical report, W3C (2004), http://www.w3.org/TR/ws-arch/
13. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1 (March 2001)
14. Cooper, G., DiPippo, L., Esibov, L., Ginis, R., Johnston, R., Kortman, P., Krupp, P., Mauer, J., Squadrito, M., Thurasignham, B., Wohlever, S., Wolfe, V.: Real-time corba development at mitre, nrad, tripacific and uri. In: Proceedings of the Workshop on Middleware for Real-Time Systems and Services, San Francisco, CA (1997)
15. Denning, P.J.: Thrashing: Its causes and prevention. In: Proceedings AFIPS, Fall Joint Computer Conference (1968)
16. Heiss, H.-U., Wagner, R.: Adaptive load control in transaction processing systems. In: VLDB 1991: Proceedings of the 17th International Conference on Very Large Data Bases, pp. 47–54. Morgan Kaufmann Publishers Inc., San Francisco (1991)
17. Leung, J.Y.-T. (ed.): Handbook of Scheduling: Alogorithms, Models, and Performance Analysis. Chapman & Hall/CRC, Boca Raton (2004)
18. Levy, R., Nagarajarao, J., Pacifici, G., Spreitzer, A., Tantawi, A., Youssef, A.: Performance management for cluster based web services. IEEE Journal on Selected Areas in Communications, 247–261 (2005)
19. Lin, B., Dinda, P.A.: Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In: SC 2005: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. IEEE Computer Society, Washington (2005)
20. Liu, Z., Niclausse, N., Jalpa-Villanueva, C.: Traffic model and performance evaluation of web servers. Perform. Eval. 46(2-3), 77–100 (2001)
21. Ludwig, H.: Web services qos: External slas and internal policies - or: How do we deliver what we promise? In: Proc. 4th IEEE Int'l Conf Web Information System Eng. Workshops, pp. 115–120. IEEE CS Press, Los Alamitos (2003)
22. Mitra, N.: Soap version 1.2 part 0.

23. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers 39(9), 1175–1185 (1990)
24. Siddhartha, P., Ganesan, R., Sengupta, S.: Smartware - a management infrastructure for web services. In: Proc. of the 1st Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI 2003), Angers, France, April 2003, pp. 42–49. ICEIS Press (2003), In conjunction with ICEIS 2003
25. Song, L., Marin, G.A.: Generating realistic network traffic for security experiments. In: SoutheastCon, 2004. Proceedings, pp. 200–207. IEEE, Los Alamitos (2004)
26. Uhlig, S., Bonaventure, O.: Understanding the long-term self-similarity of internet traffic. In: Smirnov, M., Crowcroft, J., Roberts, J., Boavida, F. (eds.) QofIS 2001. LNCS, vol. 2156, p. 286. Springer, Heidelberg (2001)
27. Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: An analytical model for multi-tier internet services and its applications. In: SIGMETRICS 2005: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, vol. 33, pp. 291–302. ACM Press, New York (2005)
28. Waldspurger, C.A., Weihl, W.E.: Lottery scheduling: Flexible proportional-share resource management. In: Operating Systems Design and Implementation, pp. 1–11 (1994)