Cloud Support for Latency-Sensitive Telephony Applications

Jong Yul Kim and Henning Schulzrinne
Computer Science Department
Columbia University
New York, NY
{jyk,hgs}@cs.columbia.edu

***Abstract*—Cloud computing is great for scaling applications but the latency in a guest VM can be unpredictable due to resource contention between neighbors. For telephony applications, which are latency-sensitive, we propose a system to monitor telephony server latencies and adapt the server load based on the measured latencies. We implemented the system and evaluated it on an Amazon EC2 testbed. We show indirectly by comparing our server on EC2 and on a local VM, that there may be contention between EC2 VMs in the wild that leads to higher server latency. While there is some overhead due to constant monitoring of the server, our system manages to lower latency by reducing the load to the server.**

I. Introduction

Cloud computing enables applications to scale in real time through on-demand provisioning of resources. Developers building dynamically scalable websites can find a wide range of scalability-support services offered by cloud providers. For example, Amazon EC2 [1] provides CloudWatch service with Auto Scale to monitor the status of web servers and automatically add or remove servers. Also, Elastic Load Balancer distributes incoming HTTP traffic to web servers while keeping track of the number of available servers. These support services aid considerably in the development of scalable websites.

In addition to websites, on-demand scalability would also greatly benefit telephony applications. Telephony traffic patterns, both time-based (e.g., diurnal and seasonal patterns) and event-based (e.g., natural disaster or televoting) [2], are conducive to automatic scaling such as those provided by Amazon. However, telephony applications have an important requirement to maintain low processing latency for call setup and voice delivery.

In a dedicated server environment, the telephony application is profiled with a fixed configuration of servers. After the configuration is proven to satisfy latency and estimated throughput requirements, the system is generally left alone except for maintenance operations. As long as load is distributed evenly among individual servers so that they are not overloaded, each server is guaranteed to meet latency requirements.

On the other hand, in the cloud environment, latency profiling and planning is considerably more complex. Virtual Machine (VM) colocation leads to resource contention within the physical hardware, resulting in significant degradation in latency [3]. In a cloud, even a normally loaded server can show increased latency due to the activities of neighboring VMs. Traditional methods of round-robin distribution and least-work distribution, therefore, may not be good for latency-sensitive applications in the cloud.

Our approach is to introduce a distributed, latency-based mechanism for load distribution within the cloud. In our design, the cloud constantly monitors each server for request-to-response latency and increases server load as long as the latency is within bounds. If the latency moves out of bounds for any reason, our system quickly reduces the load so that other servers with good latency can process the incoming load. A server will rebound to increased load if latency improves. Since each server is essentially requesting load based on available capacity, centralized load balancing is no longer necessary. Instead, there is a *front-end queue* from which all servers pull their workload.

We tested our approach using a set of Session Initiation Protocol (SIP) proxy servers and measured the performance on Amazon EC2. SIP proxy servers relay call establishment and teardown requests between callers. Delays in SIP proxy servers result in long call setup time. These SIP servers scale well because each call is handled by a unique server and there are no dependencies between servers.

While we designed and implemented our approach with the telephony application in mind, there are many other latency-sensitive applications for which our approach may be useful, such as stock trading systems and airline reservation systems where the request-to-response latency is measured in at least hundreds of microseconds. The server monitor and the front-end queue are abstract enough to be implemented by cloud providers and configured at deployment to support these types of applications.

Our contributions are:

- Introduction of a distributed, latency-based load distribution policy for telephony applications in the cloud.
- Identification of new components in the cloud that can assist latency-sensitive applications.
- Measurement and evaluation of our policy on a commercial-grade cloud platform (Amazon EC2).

The rest of the paper is organized as follows. Related work is explained in Section II. Section III goes over a little bit of background in SIP. We explain the design of our latency-sensitive application support service in Section IV. Our particular implementation of the system is described next in

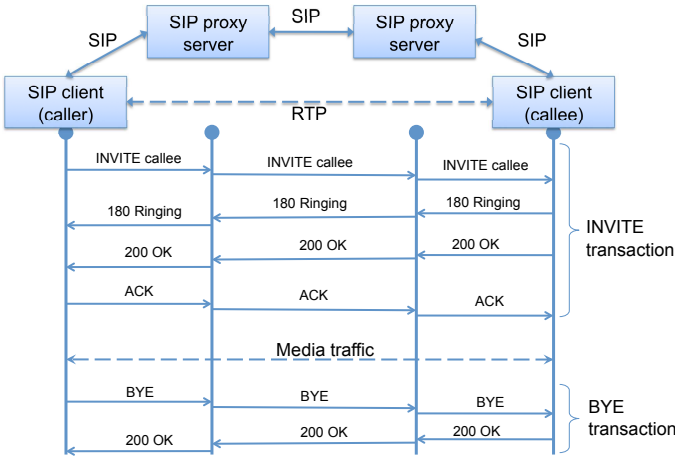

Fig. 1: The SIP trapezoid shows how packets travel between callers. The INVITE transaction establishes a call and the BYE transaction tears it down.

Section V. Section VI is a performance evaluation of various aspects of the system. Lastly, Section VII concludes the paper.

II. Related Work

There are several load distribution mechanisms used in the cloud. Round-robin seems to be the most popular while some cloud providers offer more advanced logic such as least-connect [4]. These load distribution mechanisms assume that latency requirement will be met as long as none of the servers are overloaded. The assumption is not valid on a cloud platform. As shown in Figure 5 in Section VI, server latency is unpredictable even under normal load, possibly due to resource contention with other virtual machines in the same physical host. Accordingly, we propose to use latency as the primary metric for load distribution when the application is latency-sensitive.

Latency on cloud computing platforms has been studied at different levels of the cloud architecture. For example, latency degradation due to resource contention between VMs colocated on the same hardware has been studied by [3][5][6]. These studies conclude that improvements in VM resource scheduling are needed so that less contention leads to better performance. These solutions attack the latency problem at the source while our solution attempts to mask the problem temporarily.

Smart VM placement [7] is another potential solution. In this approach, the VMs are assigned to hardware depending on application workload to minimize interference. However, there is a problem: the cloud provider does not know the application workload and the application developer has no power to decide where to place their VMs. VM placement is useful, nonetheless, for private clouds where both the cloud provider and the application developer can share information about application workload.

For real-time applications with requirements to optimize latency at a few microsecond range, the hardware, hypervisor, and guest operating systems must be tuned [8]. But a platform

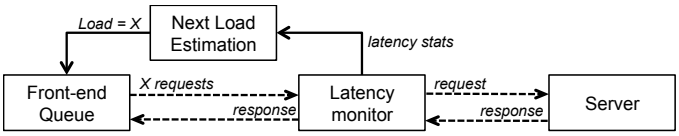

Fig. 2: Overall design of cloud support for latency-sensitive telephony application

highly optimized for real-time applications leads to degradation of performance in other applications [8], and is not an appropriate solution for a general-purpose cloud platform.

Lastly, there is a lot of interest in deploying massively multiplayer games on the cloud [9][10]. Games are latency-sensitive as well; otherwise, the gaming experience degrades to an intolerable level. However, games have different traffic characteristics from telephony applications.

III. A Quick SIP Background

The Session Initiation Protocol (SIP) [11] is an IETF standard for Internet telephony. SIP is used to establish, modify, and tear down sessions. SIP messages are text-based and are syntactically similar to HTTP messages. Messages beginning with verbs like INVITE are used to establish a call and BYE to tear it down. Responses have status codes such as "200 OK" if the callee picked up the phone, or "486 Busy here" if the callee is currently busy taking another call.

SIP servers are divided into three classes with respect to the amount of call state stored: stateless, transaction-stateful, and session-stateful. Stateless servers do not store any call state and merely forward messages based on their content. On the other extreme, session-stateful servers store call state from beginning to end of a call. These servers are used for accounting and billing. Transcation-stateful servers create and store call state only until the end of each transaction.

As shown in Figure 1, a simple call consists of two transactions: an INVITE transaction and a BYE transaction. An INVITE message from the caller initiates an INVITE transaction and an ACK message terminates the INVITE transaction. If a proxy server stores transaction state, all messages within this transaction must be processed by the same proxy server. SIP headers contain information such as the Call-ID to match individual messages to transactions and sessions. In a server cluster, the Call-ID is important in assisting load balancers to send messages to the right server.

IV. Design

Figure 2 shows the design of the cloud support system for latency-sensitive telephony applications. It is a feedback loop that consists of three functions: monitoring latency on a server, estimating the next load, and distributing load from the front-end queue. In this design, latency is constantly monitored and used as input to estimate how much the next load should be. If the latency increases, next load is reduced. If latency stays stable or decreases, a higher load is tried. In short, load offered to the server is dependent on currrent latency statistics. Periodic updates of the load are sent to the front-end queue. Since each server is essentially determining its own load in real

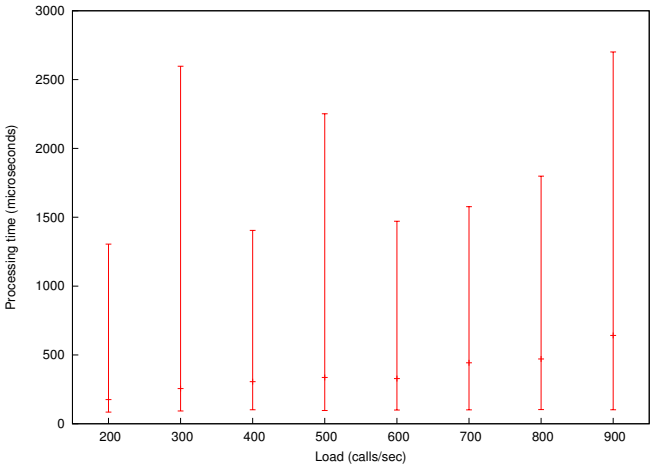

Fig. 3: Load versus latency of a SIP proxy server. Each vertical line shows the minimum, 95th percentile, and maximum latency.

time, the role of the front-end queue is to passively distribute load to each server, up to the amount requested.

Placing a queue in the front may be counter-intuitive in a system that supports latency-sensitive applications. Indeed, queuing delay can rapidly increase if traffic arrival rate is greater than the server processing rate. However, by centralizing the likelihood of queueing delay to the front-end queue, we can better manage the delay. The best option is to have a queueing delay close to zero. This close-to-zero queueing delay is guaranteed by making sure that the sum of all server capacities is greater than the traffic arrival rate, a strategy that is feasible with dynamic scalability of cloud computing platforms. The other option is to reduce the queue length to control delay. Unfortunately, this results in dropped packets and lost calls but the upside is that latency can be managed for the calls that got through.

Different aspects of the design are explained next.

A. Monitoring latency

As shown in Figure 2, the latency of a server is monitored by tracking request-response pairs. The request-response pair is application specific.

For SIP proxy servers, the request is an incoming SIP INVITE message, which starts an INVITE transaction, and the response is a proxy-generated SIP INVITE message to the callee. The pair is matched with a Call-ID header value within the SIP message since Call-ID is not modified by the proxy server. The time difference between the request-response pair measures the sum of latencies involved in receiving and parsing the SIP INVITE message, generating and storing transaction state in memory, querying the database for callee location, possibly writing the call record to database for accounting purposes, generating a SIP INVITE message to send to the callee, and finally sending the message.

Thus, the measured latency is not only the processing latency of the server, but the total latency of the flow of execution involving all participating nodes in the server cluster. For example, if the database is suddenly slow, server latency measurement is affected as well. This will result in decreased load for the server even though the server has room for more load. However, decrease in server load will lead to a decrease in database queries, which may relieve load on the database and return latency back to normal.

The on-going measurement is used to generate latency statistics such as the minimum, maximum, median, the 95th percentile value, average, and standard deviation. For each time window, e.g., 1 second, these statistics are gathered and used as input to calculate load appropriate for the server.

B. Estimating load

Load sent to the server directly affects the server's latency. Figure 3 shows our measurement of a SIP proxy server's load and latency. At a given load, each vertical line shows the value of three statistics: minimum, 95th percentile, and maximum. The minimum value remains constant across all loads, since this is the latency when there is no resource contention. The maximum value is the worst-case latency that is affected by CPU scheduling, cache misses, memory page faults, resource contention, and network delays. To guarantee a strict constraint on maximum latency, one must work from the bottom-up starting with hardware and operating systems that respond within a given timeframe. This is difficult and expensive, so most applications use a percentile value such as the 95th percentile value.

By controlling load on the server, the 95th percentile latency can also be controlled. Therefore, when an external event such as VM contention starts to degrade latency, the server reduces its load so that the 95th percentile latency is still within limits.

However, care must be taken not to overestimate the load on the server. If the load is overestimated, latency will degrade very quickly. Placing a hard limit on the maximum load helps to keep the latency reasonable in normal situations without VM contention. During periods where latency is high even when the load is below the maximum, we use a conservative policy when increasing the load on the server so that load is increased at 10 calls every minute. When reducing load, we reduce to 90% of the current load as soon as the 95th percentile latency moves out of bounds.

The estimated load is updated periodically to the front-end queue. The protocol is very simple. An update message is sent every x seconds to the front-end queue. The message contains the IP address of the server and the server capacity. The front-end replies with an OK message. A lost update message can degrade the overall performance of the system. Therefore, TCP is used as the underlying transport protocol to prevent message loss. Since update messages are short and throughput is not a concern, TCP congestion avoidance or flow control algorithms are irrelevant for this purpose.

C. Front-end queue

The behavior of front-end queue is described in pseudocode in Listing 1. Thread 1 receives an incoming message and checks to see if there is a {Call-ID, server} pair in memory. If there is one, then the message is sent directly to the server. This

is known as sticky session and is important for correct behavior of SIP servers. As explained in Section III, SIP messages are bound to the session state in a server. SIP proxy servers will reject messages that do not belong to a session, unless it is a message that starts a new session. If the {Call-ID, server} mapping does not exist and the message is a SIP INVITE message, it is stored in queue.

Thread 2 takes the first message from queue and chooses a destination server. This creates a new {Call-ID, server} mapping in memory so that subsequent messages will be correctly routed to the server. Once it sends the message, server capacity is decreased by one.

```
1  MSG: incoming message
2  Call-ID: Call-ID value in message
3  SERVER: a proxy server
4
5  Thread 1:
6    receive incoming MSG
7    parse Call-ID from MSG
8    if {Call-ID, SERVER} mapping exists
9      send MSG to SERVER
10     if MSG is session-ending message
11       remove {MSG, SERVER} from mapping
12   else
13     store MSG in queue
14
15 Thread 2:
16   take the first MSG from queue
17   choose a SERVER with capacity > 0
18   add {Call-ID, SERVER} to mapping
19   decrement SERVER capacity by one
20   send MSG to SERVER
```

Listing 1: Front-end algorithm

The queue length is one of the factors in deciding queueing delay. We use a varying queue length to control how long a packet waits in the queue.

D. Server scaling logic

From the front-end queue, the traffic arrival rate and the total server capacity can be obtained. A server needs to be added when the traffic arrival rate λ approaches the total server capacity θ, and removed when the total server capacity is less than the capacity of $n-1$ servers. We use a simple 80% threshold on the $\frac{\lambda}{\theta}$ ratio to add a server.

The bigger problem is the startup time of VMs in the cloud. On Amazon EC2, it takes around 1 to 2 minutes on average to start a new VM [12]. To offset this starting time, a spare VM can be powered on in advance. This type of $n+1$ server configuration will not be needed with improvements in startup time.

To remove a server, we follow the index-packing scheme [13] which distributes load to a small number of servers so that idle servers can be turned off after a timeout. SIP proxy servers are usually transaction-stateful at most; therefore, an idle server can be turned off after a timeout of few minutes when all transaction states within the server expires.

The total server capacity can fluctuate from time to time because each server's load is determined by individual latency statistics. Under normal conditions, the total server capacity would remain stable because latency fluctuations in a small set of servers would not affect the overall capacity. However, if there is a cluster-wide disruption in latency, the total server capacity may drop significantly. The problem, then, is not with the latency-support system but a much bigger problem involving the telephony application or the underlying cloud computing infrastructure.

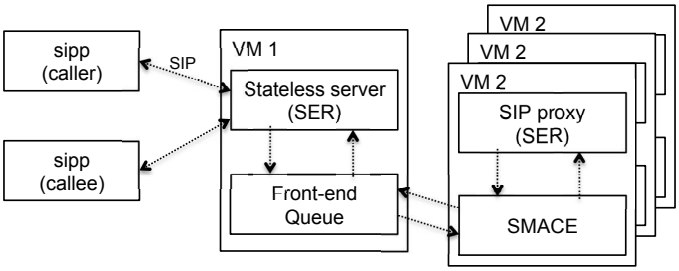

Fig. 4: Experiment setup on Amazon EC2. Cassandra DB, used for storing user location and credentials, is not shown in the figure.

TABLE I: Type of EC2 instances used for evaluation. VM 1, VM 2, and sipp are shown in Figure 4

Component	VM Type	Cores	Memory	Network
VM 1	c1.medium	2	1.7 GiB	Moderate
VM 2	m1.small	1	1.7 GiB	Low
sipp	m1.small	1	1.7 GiB	Low

V. Implementation

We implemented two prototype components to test our approach. The front-end queue (FEQ) is a component described in the previous section and the Server Monitor And Capacity Estimation (SMACE) is a component that monitors servers and estimates capacity. Both components are written in C for fast execution time. Figure 4 shows the location of these two components in the server cluster.

A. Implementation of SMACE

SMACE measures the proxy server's request-response latency by passively sniffing messages using libpcap [14]. This incurs some overhead and an evaluation of throughput and latency overhead is presented in Section VI. We used online algorithms to generate statistics like the average and standard deviation, and the P2 algorithm [15] for estimating the median. Minimum and maximum are trivial to obtain. For 95th percentile value, we implemented a small, sorted array that is used to keep track of the top 0.05% of the values we have seen in a time period. The smallest value in the array is the 95th percentile value. This method of calculating the 95th percentile value is not scalable but it is still useful and fast enough for our servers' peak load of less than 500 calls per second, which translates to at most 25 values in the array.

The user sets the limit on the 95th percentile latency and SMACE will adjust load in order to meet those guidelines. To eliminate overestimation of load, we introduced a user-configurable parameter to specify the maximum load of the server. This value can be easily obtained from a stress test of a server. Every second, SMACE updates the server capacity to FEQ.

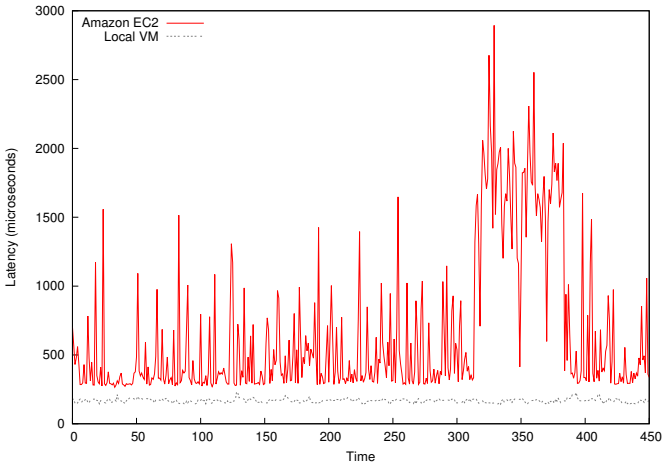

Fig. 5: 95th percentile latency on Amazon EC2 versus local VM. Server on Amazon EC2 displays much more variation than the server on a local VM at the same low load level of 300 calls/sec.

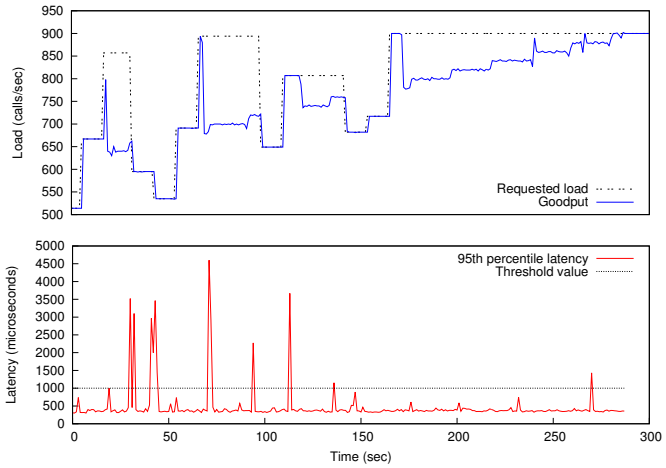

Fig. 6: SMACE tries to bring latency down by reducing load when latency is above threshold.

TABLE II: Comparison of Amazon EC2 and local VM

	Cores	CPU Model	Memory	Kernel
EC2 m1.small	1	Xeon 5110 (1.60GHz)	1.7 GiB	2.6.35.14
Local VM	1	Xeon E5-2650 (2.00GHz)	1.0 GiB	2.6.27.21

B. Implementation of FEQ

The Front-End Queue (FEQ) passively distributes messages to proxy servers, up to the load requested by each server. FEQ only queues messages without an associated session. Messages within existing sessions are forwarded to the appropriate server immediately. The queue is allocated in memory with a fixed size, but a variable called *queue_length* is used to control the effective size of the queue. When *queue_length* is filled, new calls are dropped. As mentioned in Section IV, however, the best strategy is to proactively spawn servers so that new calls can also be processed.

VI. Evaluation

A. Evaluation setup

Most of our experiments were conducted on Amazon EC2. Figure 4 shows our testbed setup. A stateless SIP server is placed in the front to deal with client interaction. Front-end queue (FEQ) interacts with SMACE to send requests to servers. An open-source SIP server called SIP Express Router [16] is used for both the stateless SIP server and the SIP proxy server. SIP load generator called sipp [17] is used for calling and receiving calls. The type of VM used on Amazon EC2 is organized in Table I.

B. EC2 versus local VM

To verify the resource contention problem in the wild, we indirectly compared the latency of a server on Amazon EC2 to the latency of the same server on a local VM. We tried to configure the local VM so that it would be very similar to the image on Amazon EC2. Table II shows our setup.

As shown in Figure 5, there is much more variation in latency on Amazon EC2. Especially from time 300 to around 400, there is a period of higher latency even though no other CPU or I/O intensive process was running within the VM. This points to the possibility of VM contention.

We verified that similar latency variations can be seen in a local VM as well, when there is contention among VMs with a single network card. As a proxy server is not CPU-intensive and moderately network-intensive, the higher latency strongly suggests that a neighboring VM on Amazon EC2 was sending or receiving packets from time 300 to 400.

C. Load estimation behavior

Figure 6 shows how SMACE reacts to variations in latency by increasing or reducing the load. The actual load, or goodput, is shown as a solid line and the requested load is shown as the dotted line. If the latency spikes above the threshold, SMACE reduces the load to 90% of the current goodput. SMACE increases requested load by a simple formula when latency is below threshold.

$$load_{next} = load_{current} \times \sqrt{1 + \frac{threshold - latency_{95}}{threshold}}$$

Due to the latency-to-threshold ratio in the formula, load increase is small if latency is close to the threshold while load increase is big if latency is safely under the threshold.

To guard against excessive fluctuations in requested load, SMACE only changes the requested load every 10 seconds. The graph shows this behavior; at around 90 and 270 seconds, requested load stays the same even though the latency spikes well above the 1000 microsecond threshold.

D. Server monitoring overhead

Server monitoring overhead is shown in Figure 7 and 8. During our measurements, goodput was not affected by SMACE although we expect minor overhead at higher loads. Figure 8 shows measurements of how long it takes for the

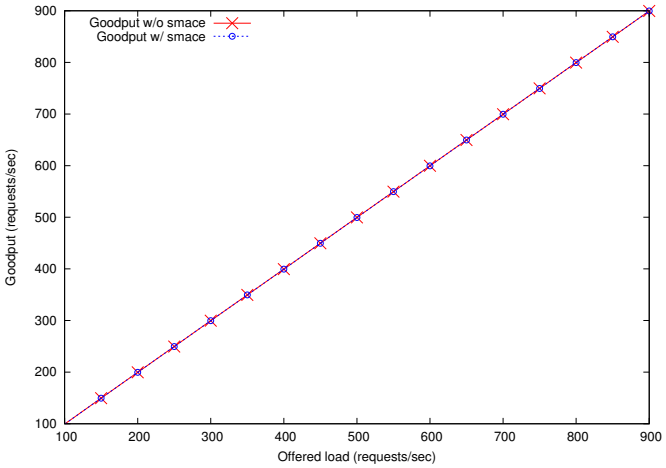

Fig. 7: Goodput versus offered load with and without server monitoring shows no degradation until 900 calls per second.

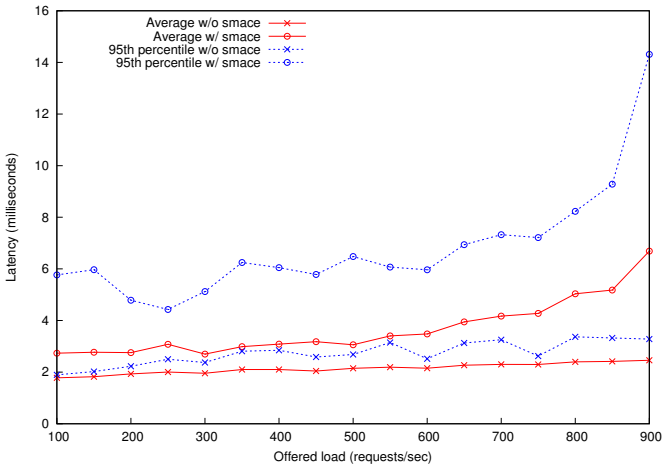

Fig. 8: Time-to-ringtone measured by sipp. Latency is measured from "INVITE sent" until "180 Ringing received".

caller to hear the ring tone. The latency shown in the graph includes network delays on both the caller and callee. Assuming a 250 millisecond round trip time to the opposite side of the globe, the ring time for international calls will still be less than 1 second even at 900 calls per second load.

VII. Conclusion

In this paper, we presented a cloud support system for latency-sensitive telephony applications. Latency variations in the cloud are unpredictable due to resource contention between VMs colocated in the same physical hardware. As a result, a server may suffer from increased latency due to resource contention even if the server load is stable.

Our system ensures that servers in the cloud are always running at peak performance with respect to latency. The server is constantly monitored for latency. Based on these latency statistics, for example the 95th percentile value, the next server load is calculated and requested from the front-end queue. The front-end queue passively distributes load to the server up to the requested amount. By reducing the load on a high latency server, the system lets other servers with better latency handle more load. When latency drops to normal levels, load is increased gradually to fully utilize the server.

Our approach was implemented for a cluster of SIP proxy servers for telephony services. This approach may also be used for other latency-sensitive applications with a request-response transaction model, such as stock trading or airline reservation systems.

Acknowledgment

This work was funded in part by an Amazon EC2 research grant. The authors would also like to thank our anonymous reviewers for their helpful comments.

References

[1] *Amazon EC2*, http://aws.amazon.com.

[2] J. Bagrow, D. Wang, and A. Barabasi, "Collective response of human populations to large-scale emergencies," *PloS one*, vol. 6, no. 3, pp. 1–8, 2011.

[3] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," *Proceedings of the first annual ACM SIGMM conference on Multimedia systems - MMSys '10*, p. 35, 2010.

[4] E. Caron, F. Desprez, L. Rodero-Merino, and A. Muresan, "Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds," in *Cloud Computing*. CRC Press, Oct. 2011, pp. 301–323.

[5] Y. Koh and R. Knauerhase, "An analysis of performance interference effects in virtual environments," *IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[6] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments," *2010 IEEE 3rd International Conference on Cloud Computing*, no. Vmm, pp. 51–58, Jul. 2010.

[7] S. Lee and R. Panigrahy, "Validating heuristics for virtual machines consolidation," *Microsoft Research Tech Report, MSR-TR-2011-9*, 2011.

[8] VMWare, "Best Practices for Performance Tuning of Workloads in vSphere VMs," *VMWare White Paper*, 2013. [Online]. Available: http://www.vmware.com/files/pdf/techpaper/VMW-Tuning-Latency-Sensitive-Workloads.pdf

[9] K. Chen, Y. Chang, and P. Tseng, "Measuring the latency of cloud gaming systems," *Proceedings of the 19th ACM international conference on Multimedia*, pp. 1269–1272, 2011.

[10] M. Claypool and D. Finkel, "Thin to win? Network performance analysis of the OnLive thin client game system," *Proceedings of the 11th ACM Network and System Support for Games (NetGames)*, 2012.

[11] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *Session Initiation Protocol*, 2002, RFC 3261.

[12] J. Kim and H. Schulzrinne, "SipCloud: dynamically scalable SIP proxies in the cloud," *Proceedings of the International Conference on Principles, Systems and Applications of IP Telecommunications*, 2011.

[13] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. a. Kozuch, "AutoScale," *ACM Transactions on Computer Systems*, vol. 30, no. 4, pp. 1–26, Nov. 2012.

[14] *Tcpdump and libpcap*, http://www.tcpdump.org.

[15] R. Jain and I. Chlamtac, "The P2 algorithm for dynamic calculation of quantiles and histograms without storing observations," *Communications of the ACM*, vol. 28, pp. 1076–1085, 1985.

[16] *The SIP Router Project*, http://sip-router.org/.

[17] *Sipp*, http://sipp.sourceforge.net/.