SipCloud: Dynamically Scalable SIP Proxies in the Cloud

Jong Yul Kim
Computer Science Dept
Columbia University, USA
jyk@cs.columbia.edu

Henning Schulzrinne
Computer Science Dept
Columbia University, USA
hgs@cs.columbia.edu

ABSTRACT

One of the features of cloud computing platforms is the ability to scale applications dynamically. Generally, this feature is used in a web services context where the web service provider adds more web servers during times of high traffic and remove web servers during time of low traffic. Real-time communications service providers can also benefit from such feature. In this project, we propose and implement a highly scalable SIP proxy architecture that utilizes dynamic scalability. An evaluation of dynamic scalability of a part of the system is presented as well.

1. INTRODUCTION

One of the advantages of cloud computing is dynamic scaling. Dynamic scaling allows cloud users to automatically expand or reduce the number of VM instances as required by the application.

This property of cloud computing is attractive for voice service providers because voice usage is predictable to some extent, e.g. showing a diurnal or seasonal patterns. A voice service provider may save costs by scaling up servers as much as needed during busy holidays and then freeing server resources for other uses during non-holiday periods. Also, sometimes there are unexpected sudden spikes in voice usage, for example, when there is a major disaster. Dynamic scaling on a cloud computing platform has the potential to handle both cases economically. It allows them to maintain only as much VM instances based on call load, freeing server resources for other uses. And in certain times, especially during large scale emergencies, it handles sudden load spikes.

To take advantage of dynamic scaling in an IP telephony context, we first present a SIP proxy architecture used by our SipCloud system that is highly scalable. The architecture is designed in such a way that individual tiers are separately scalable from others and also highly scalable. To maximize scalability, we used a NoSQL key-value store called Cassandra [3], which is becoming increasing popular as a scalable web backend. We discuss the challenges in using a non-relational distributed database for a SIP proxy system.

For an application such as a SIP proxy system to optimize its scaling dynamics, the cloud platform must support application-level load monitoring and scaling. However, commerical cloud providers currently provide load monitoring that are restricted to lower layer metrics such as CPU load, memory usage, network bandwidth, and disk usage and latency. Amazon supports an application level monitoring and scaling via a feature called Elastic Load Balancers [2] but this only works for HTTP traffic.

Lastly, we evaluate dynamic scalability of a part of the system on Amazon EC2 and present the results.

Our contributions are:

- We designed and implemented a highly scalable SIP proxy architecture that takes advantage of dynamic scalability offered by cloud computing platforms.
- In the process, we modified SIP proxy with a non-relational distributed DB backend called Cassandra. Our experience with a non-traditional database integration is discussed.
- We built and evaluated a part of the system for dynamically scaling SIP proxies on Amazon EC2.

2. SIPCLOUD ARCHITECTURE

To fully utilize the dynamic scaling properties of Infrastructure-as-a-Service platforms, the architecture of the real-time communications system that runs on top of the cloud must be adaptive to load dynamics and highly scalable. In this section, we present an architecture of such a system and explain the reasons that makes it adaptive and highly scalable.

The SipCloud architecture is shown in Figure 1. A DNS server is outside the cloud and directs clients to one of the load balancers in the cloud. Inside the cloud is the classic three-tier architecture that consists of the load balancer tier, the SIP proxy server tier, and the database server tier. Load balancers distribute incoming SIP messages among available SIP proxy servers, SIP proxy servers process messages, and the database servers store user registration information needed by the proxy servers to process messages. Each component in the three tiers run on an individual virtual machine instance.

All of the components in the SipCloud system are orchestrated by a centralized control entity called the Load Scaling Manager (LSM). The LSM monitors load and takes action

IPTComm '11, August 01 - August 02 2011, Chicago, IL, USA
Copyright 2011 ACM 978-1-4503-0975-2/11/08 ...$10.00.

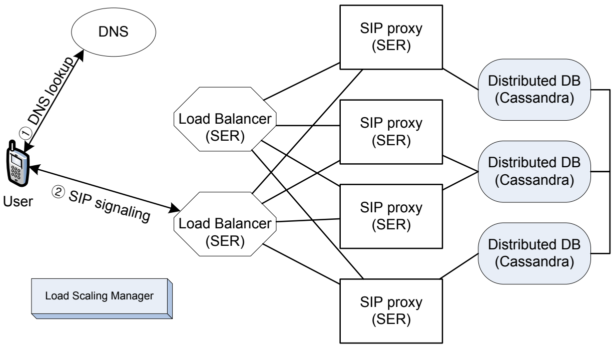

Figure 1: SipCloud Architecture

such as adding or removing a component to a particular tier. The LSM is described in detail in Section 3.

In the SipCloud system, each tier is *highly scalable* and *independently scalable*. These two properties allow the system to be adaptive and highly scalable.

2.1 Highly Scalable Tier

A highly scalable tier means that the tier can have an unbounded number of components. A tier's high scalability is achieved by using components that can function without synchronizing with other components in the same tier. This means that the full functionality of individuals component does not depend on other components in the same tier. This property allows the LSM to freely add new components to a tier.

To this end, the load balancer tier in the SipCloud architecture use an identifier-based hash on incoming SIP messages to distribute load to SIP proxies. Because it is using a hash function, there is no need to synchronize the load balancers with each other. These load balancers can be added or removed by the LSM without affecting other load balancers.

Similarly, in the SIP proxy tier, SIP proxies run independently from other SIP proxies. Due to identifier-based hashing of SIP messages at the load balancer tier, each SIP proxy handles its own set of calls.

For the database tier, components cannot be fully independent since data is replicated among the components to ensure availability. This requirement of data availability limits independence of database components, thus making the tier the least scalable compared to the load balancer tier and the SIP proxy tier. In this case, one mechanism high scalability can be achieved is by using components that organize into a peer-to-peer group and that use a hash table to distribute data among the components. An example of such a peer-to-peer distributed database server is Cassandra [3]. SipCloud uses Cassandra nodes as components of its database tier. Further discussions about using Cassandra for SIP proxies is presented in Section 2.4.

2.2 Independently Scalable Tiers

Each tier performs tasks that scale differently from other tiers. For example, the database tier stores user registration data and needs to scale up when the *number of subscribers* increase. On the other hand, the SIP proxy server tier and the load balancer tier needs to scale up when the *number of calls* increase. Load balancer tier and SIP proxy server tier scale separately due to the difference in message processing throughput. Generally, SIP proxy server tier needs to scale up faster than the load balancer tier.

SQL Query		Cassandra Query	
Command	`SELECT`	Method	`get`
Table	`credentials`	ColumnFamily	`credentials`
Columns	`password`	Columns	`password`
Criteria	`userID=zebra`	Key	`zebra`

Table 1: Mapping an SQL query to a Cassadra method call

In a system with independently scalable tiers, components can be added to or removed from each tier as needed by the tier. Because there's no dependency on other tiers, each tier can scale up or down by its own mechanism. This has a good side effect in that each tier can self-scale according to the requirements of the tier. Therefore, the scaling logic is simplified to a tier-local decision. Therefore, the LSM is able to quickly decide when and how to scale each tier.

Overall, this has the potential to shift the bottleneck of the system from one tier to another. For example, as the bottleneck of the load balancer tier is solved by adding more load balancers, another tier may become the new bottleneck. However, since all tiers are separately scalable, each bottleneck is removed adaptively.

2.3 Implementation

As stated earlier, each element in the three-tier architecture runs as a separate VM instance in the cloud. For example, in Figure 1, there are two load balancer instances, four SIP proxy server instances, and three distributed database server instances. The minimum functional requirement is that there is always one VM instance running in each tier.

For the load balancer tier and the SIP proxy server tier, the system uses an open-source SIP server called SIP Express Router (SER) [4]. SER runs as a full SIP proxy server in the SIP proxy tier but runs as a SIP-level load balancer in the load balancer tier. Each SER process runs independently from other SER processes.

To implement the data storage tier as a highly scalable tier, the SipCloud system uses the Cassandra distributed database server [3]. Cassandra is a P2P-based key-value store which is used by Facebook, Twitter, and Digg. Its scalability and self-organizing properties are the main reason it was chosen.

The three tiers are independently scalable: components in each tier does not affect the scalability of another tier. The only dependence between tiers is for configuration. For example, a SIP proxy process needs to know which Cassandra process to contact to store or retrieve data. A load balancer needs to know how many SIP proxies there are to send load to the proxies. But this does not affect scalability.

2.4 SIP data model for Cassandra

Cassandra was chosen because of its scalability and self-organizing properties. However, unlike traditional databases, Cassandra is a key-value store that does not support structured query languages (SQL). This was one challenge in implementing a database driver for the SIP proxy.

A basic SQL query has four parts: a command, a table to retrieve values from, a list of columns within the table, and a list of criteria to match. For example, a SELECT query looks likes this:

```
SELECT name, contact FROM location
                      WHERE userID="zebra"
```

The mapping from an SQL query to a Cassandra query is shown in Table 1. For a SIP proxy to use Cassandra, these four parts must have a corresponding mapping to a key-value pair: an SQL command maps to a Cassandra command, a table maps to a Cassandra ColumnFamily, the list of criteria maps to the key, and the list of columns maps to the values.

There are some SQL queries that do not map one-to-one to a Cassandra query. For example,

- when there are more than one criteria in the SQL query, key has to be created by concatenating the criteria, e.g., WHERE userID = “zebra” AND domain = “cs.columbia.edu” maps to key = “zebra@cs.columbia.edu”.

- * denotes all columns in a SQL query. This maps to a different method call in Cassandra called `get_slice`. `slicePredicate` is an argument to the method call that limits how many columns to fetch, by range or by count. In fact, any multi-column retrieval maps to the `get_slice` method call.

For the two tables that we used for the project, namely the `location` table which stores contact information and the `credentials` table which store user ID and passwords, these challenges were solved by using the heuristics mentioned above.

However, a query like `DELETE * FROM location WHERE expiration > 50000` cannot be mapped to a Cassandra query. In a key-value store like Cassandra, a criteria with comparisons other than equality is not straight-forward to implement.

3. DYNAMIC SCALING

Dynamic scaling involves close monitoring of system load, VM instance creation or termination, and configuration of VM instances. In the SipCloud system, these tasks are performed by an entity called the *Load Scaling Manager (LSM)*.

As shown in Figure 2, the LSM monitors loads of the proxy tier and the load balancer tier, creates or terminates VM instances for all three tiers, and (re)configures the components as the system scales up or down. For the distributed DB tier, scaling decision is made manually by a human operator and not by LSM. This is reasonable given that the distributed DB tier scales proportional to the number of subscribers which does not increase or decrease drastically in a matter of minutes. Therefore, the following description of dynamic scaling applies to the load balancer tier and the proxy tier.

3.1 Load Monitoring

The LSM monitors system load by polling each component in the load balancer tier and the SIP proxy tier for the current call load, measure in calls per second. It uses the same load monitoring mechanism for both tiers. The SIP Express Router contains modules that allow it to answer XML Remote Procedure Call (XML-RPC) queries about the component state. LSM polls each component in both tiers periodically using XML-RPC to query the current call load. Polling interval is configurable but the default is five seconds.

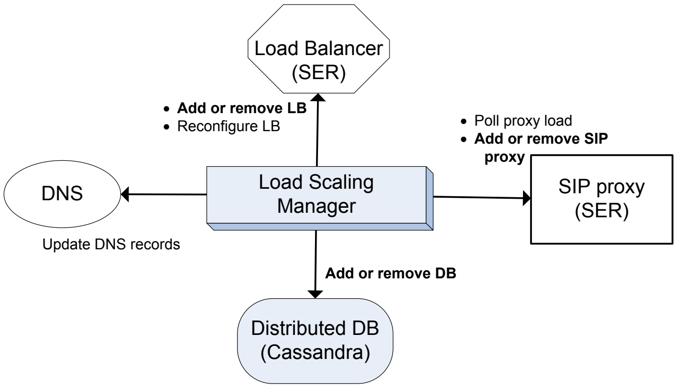

Figure 2: The Load Scaling Manager (LSM)

Periodic polling allows LSM to scale up the system before calls are dropped due to overload and scale down when it is safe to do so. Even though LSM polls individual components, the decision to scale up or down is based on the *aggregate tier load*, the sum of all call loads in the tier. When the aggreate tier load surpasses a certain threshold, LSM adds a new component to that tier in anticipation of increased load. On the other hand, when the aggregate tier load diminishes below the threshold of the tier, LSM removes a component from the tier.

Using the aggreate load to decide scaling up or down can be dangerous if the load is not evenly balanced across all components in the tier. If an individual component is overloaded, there may be calls lost even if the aggregate load on the tier is below threshold. In the SipCloud system, this is not a problem because for the SIP proxies, the load balancers use a hash function and distributes load evenly to all available SIP proxies. For the load balancers, the DNS server uses a round-robin algorithm when answering domain SRV queries.

Lower-level metrics such as CPU utilization, memory status, disk and network I/O rates, and so on can also be useful for load monitoring. While LSM does not currently use these metrics for load monitoring, it could be extended to use them to make scaling decisions.

3.2 VM creation

LSM creates and terminates VMs through the use of cloud provider APIs. VM creation takes time, generally two to three minutes or less. Therefore, a dynamically scalable system such as SipCloud must be tuned to start the process of VM creation at least two to three minutes earlier than system overload. The threshold mentioned in the previous section is one parameter that can be tuned.

3.3 Configuration

For each tier in the system, there are configurations that are needed for individual components. LSM has a global view of the whole system whereas individual components do not. Therefore, LSM is reponsible for configuring individual components as well.

For a new SIP proxy VM instance, it needs to be configured with Cassandra’s IP address so that user registration data can be retrieved. LSM accomplishes this by logging

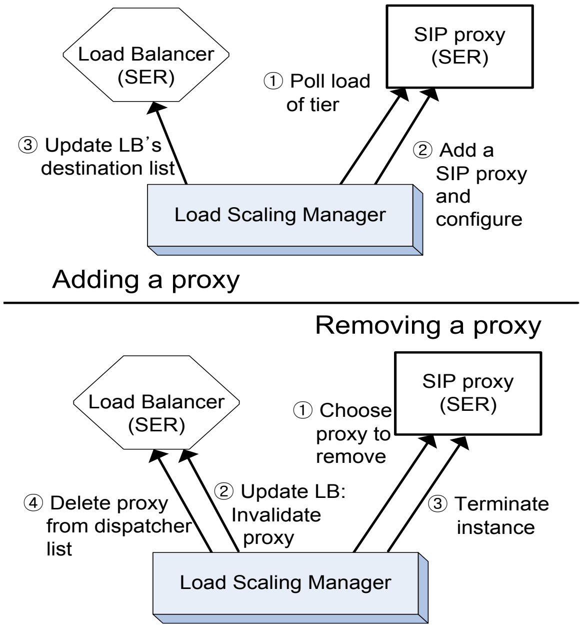

Figure 3: Adding and removing a SIP proxy

into the VM instance via SSH and executing pre-defined commands. As shown in Figure 3, once the configuration and starting of SIP proxy is done, LSM updates all the load balancer's destination list so that it includes the new SIP proxy. After reconfiguration, the load balancers start sending traffic to the new SIP proxy.

For a new load balancer VM instance, a file containing the IP address and port of all SIP proxies must be uploaded to the instance. LSM builds the file locally and uploads it to the load balancer instance using scp [1]. And it logs into the load balancer instance to start the process. When the load balancer is ready to receive calls, the LSM then updates the DNS server with a new SRV record that contains the new load balancer's public IP addrress. Once the DNS server is configured, the SRV records are sent to clients.

3.4 VM termination and reconfiguration

Removing a SIP proxy is a different process from adding one, as shown in Figure 3. If the aggregate tier load becomes lower than the threshold value, the LSM selects a SIP proxy instance to retire. Then it sends requests to the load balancers to flag the SIP proxy as *inactive*. Load balancers immediately stops sending transactions to that proxy. LSM then terminates the proxy VM instance. And on completion it updates all load balancers' destination list so that the proxy is permanently removed.

For load balancers, when the aggregate tier load becomes lower than the threshold value, the LSM requests the DNS server to remove a resource record of one of the load balancers. That load balancer needs to be available until all records sent to clients expires, so the LSM waits for the duration of the record expiration time. After this duration, the LSM proceeds to remove the VM instance from the cluster.

3.5 LSM failure

Even though LSM is a central entity in the operation of the SipCloud system, existing components and operations are not affected by LSM failure. However, the system loses its ability to scale dynamically when system load changes. When the LSM comes back online, it can resume its operations by querying the cloud about its VM instances and types.

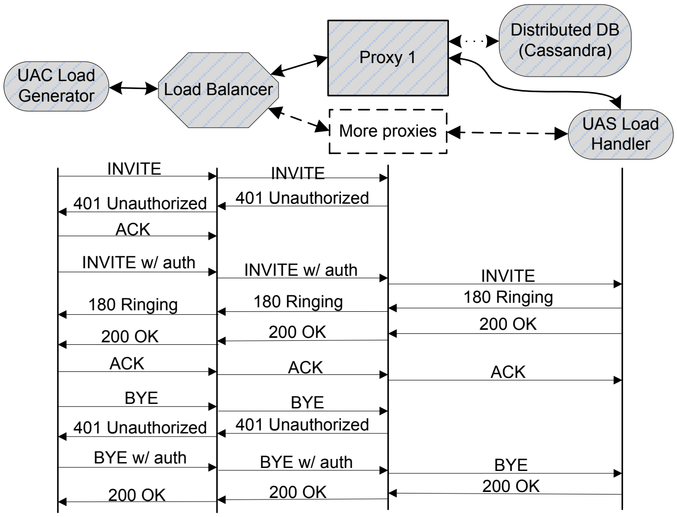

Figure 4: Test setup and message flow

4. EVALUATION

We evaluate the SipCloud system's dynamic scaling of SIP proxies on Amazon EC2 [2]. In conducting the evaluation, we had to be careful not to trigger the Amazon EC2 security alert system. Therefore, our experiments did not stress test the SipCloud system full throttle. Instead, component load tests were done to a certain degree to make sure that the load balancer, SIP proxy, and Cassandra database server are not bottlenecks in our test scenario. Then we conducted a simple, low-load test involving one load balancer and one Cassandra database server to show the dynamic scaling of SIP proxies, both in scaling up and scaling down.

4.1 Test Setup

The goal of the test is to see that dynamic scaling of SIP proxies works and to observe the behavior of the system when one load balancer and one Cassandra node is used. The test setup and the message flows are shown in Figure 4. User Agent Client (UAC) calls generated by sipp [5] goes to the load balancer which statelessly forwards calls to Proxy 1. Proxy 1 authenticates INVITE and BYE messages. In this step, there is a query to Cassandra to retrieve the digest authentication hash value. Since both the SIP proxy and the load balancer are based on the same SIP Express Router (SER) code, adding authenication to the message flow allows us to test a more realistic situation where SIP proxy incurs higher load than a load balancer. Each box in Figure 4 is a separate VM instace, including the User Agent Client (UAC) load balancer and User Agent Server (UAS) load generator. Our test was done all within the EC2 platform.

The entire message flow in Figure 4 is considered one call. Sipp considers the call a success if the message flow is followed properly from top to bottom, except for optional messages such as 100 Trying and 180 Ringing. A call is considered unsuccessful if sipp receives any unexpected message

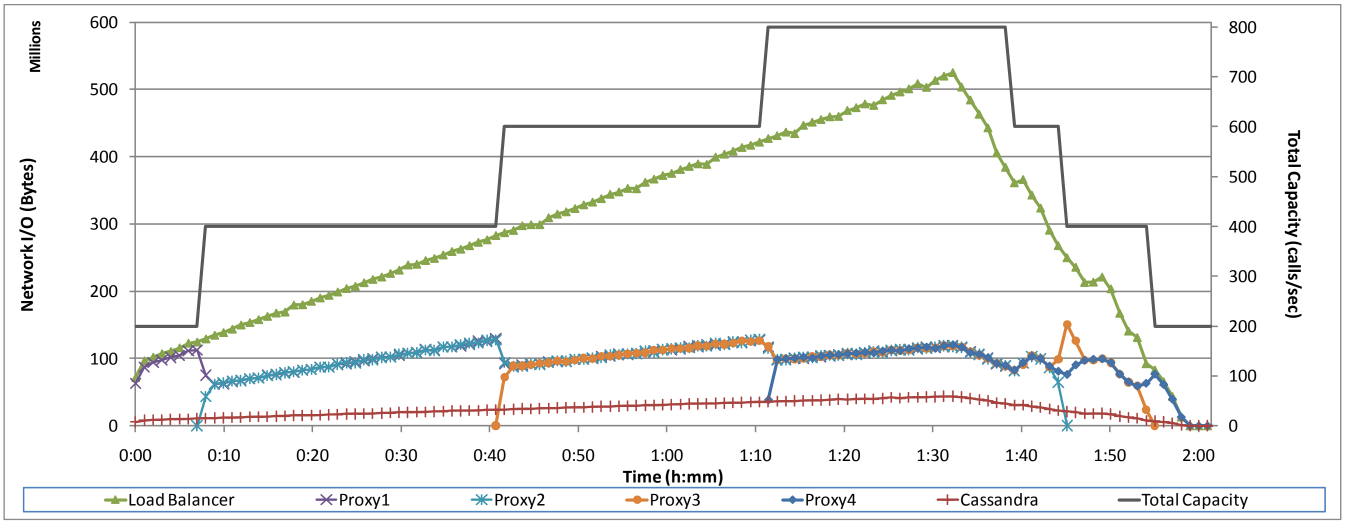

Figure 5: Total SIP proxy capacity and aggregate network I/O per VM instance

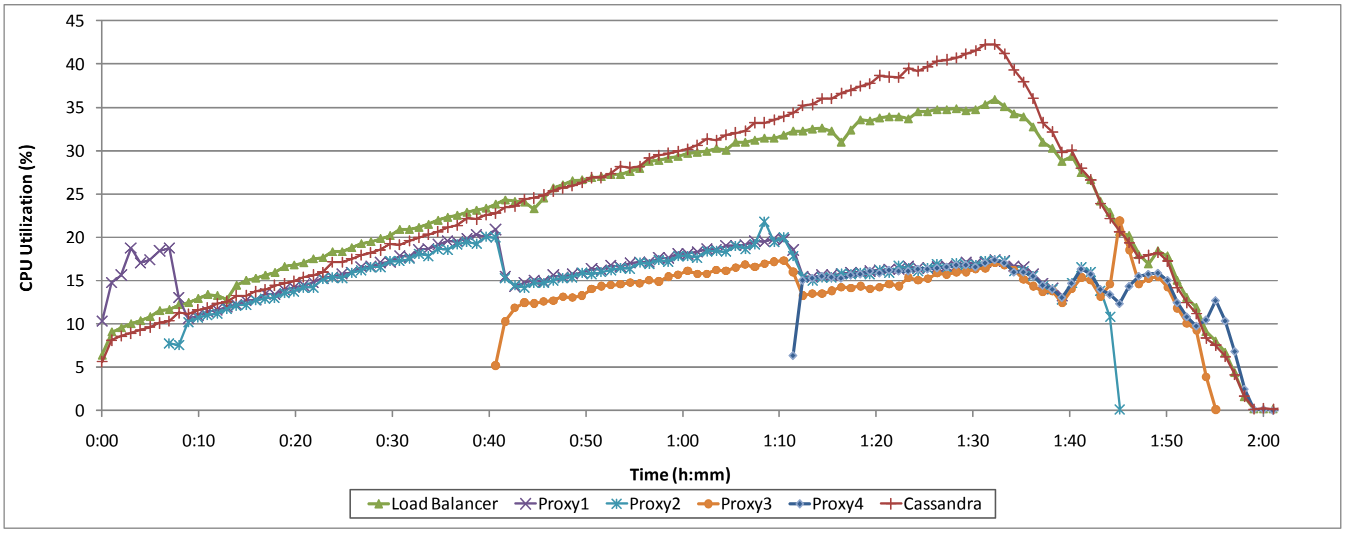

Figure 6: CPU utilization of VM instances

such as a 408 Request Timeout, or if an expected message is received out of order. A call that never completes is also a failed call. As all our test components were inside the EC2 platform, we configured sipp to use UDP without retransmission.

The M1.Large instance type was used for all components. Each of these instances have 7.5 GB of memory, a virtual dual-core processor, 850 GB of local storage, and a 64-bit linux operating system. We observed that the load balancer implementation, based on a SIP Express Router (SER), can handle more than 2300 INVITE forwards per second. The SIP proxy implementation, again based on SER, can handle up to around 520 calls per second (cps) with both INVITE and BYE authentication through Cassandra. A Cassandra node can handle more than 1800 authetication queries per second, or 900 calls per second. As mentioned before, we did not stress test the components to the limit as our tests were done within the public cloud platform. However, these numbers were referenced so that the dynamic scaling test would not suffer from bottlenecks related to either a single Cassandra node or a single load balancer.

In the test, the SIP proxy's capacity is configured at 200 cps which is much lower than 520 cps it is really capable of. But the goal is to scale up to 4 SIP proxies and then scale back to 1 SIP proxy at the end of the test. Therefore, at 200 cps for a SIP proxy, the single Cassandra node would not be overloaded even with 4 SIP proxies in the system.

Before the aggregate load on the proxy tier reaches a multiple of 200 cps, the LSM starts to launch a new proxy VM instance so that the load can be distributed without overloading any SIP proxy. The threshold is set at 85% of the proxy tier's maximum load for both scaling up and scaling down. Therefore, at total proxy loads of 170 cps, 340 cps, and 510 cps, LSM will launch or remove a SIP proxy instance. To guard against load fluctuations, the LSM only launches or removes a VM instance if the threshold is crossed for 5 consecutive monitoring sessions with 3 second intervals, resulting in a 15 second wait.

At startup, the load generated by the UAC is 140 cps. The load is increased automatically at a rate of 10 cps every 90 seconds until it reaches 760 cps. At that point, the load is reduced at a rate of 60 cps every 90 seconds until the total load reaches zero.

In gathering data for the test, we mainly used EC2 CloudWatch which allows users to monitor minute-by-minute change of various metrics such as CPU utilization and network usage in the VM instances.

4.2 Results

As it can be seen in Figures 5 and 6, the SipCloud system is able to dynamically scale SIP proxies using a cloud platform. In both figures, the X-axis shows the time from beginning of the test to the end after around 2 hours.

Figure 5 shows the total capacity of the system and the network I/O of individual VM instances. Because the two metrics have different units, there are two Y-axis in the graph. The one on the right shows total capacity in the system as proxies are spawned and terminated. Since we assumed, for the test, that SIP proxies have 200 cps maximum capacity, each SIP proxy brings 200 cps more to the total capacity whenever it is added to the system. This is shown in the graph as a step function.

The Y-axis on the left shows the sum of network input and output of VM instances. As can be seen at the top of the graph, the load balancer and Proxy 1 have similar network I/O in the beginning but soon diverges as Proxy 2 comes online. The load balancer's load is generally divided evenly among the SIP proxies due to the hash function used to distributed load. This can be observed by the fact that whenever a new proxy instance comes online, the network I/O graphs of all proxy instances merge to look like one line.

However, near the end, as proxy instances go offline due to insufficient call load, there are spikes in the network I/O of some proxy instances. This is because the load balancer distributes the load of the outgoing proxy while it is in an inactive state to one of the remaining proxies. When the outgoing is proxy is terminated, then the load balancer starts to distribute the load evenly among the remaining proxies.

The lowest aggregate network I/O is shown by the Cassandra node. The network I/O is similar to that shown by the load balancer since, like the load balancer, Cassandra is a single node in the system.

Figure 6 shows the CPU utilization of each VM instance in the system. CPU utilization for any SIP proxy instance is similar as they process same amounts of load. As for the load balancer and the Cassandra node, CPU utilization rises linearly as the load increases and both are likely to be simultaneous bottlenecks if we were to do a full throttle test on the system.

5. RELATED WORK

Kundan, et al. discusses various architectures for scalable real-time communications services [6] . In particular, they propose and evaluate a two-stage message processing architecture where a DNS server first directs traffic to the first-tier proxy, which in turn direct traffic to one of the proxy clusters in the second-iter. This is similar to our architecture in that there are three tiers after the DNS server. The difference is in the second tier and the third tier where we used independent SIP proxies and Cassandra database servers instead of proxy clusters and MySQL database. However, the main difference is that our work studies dynamic scalability of such architectures.

Dynamic scaling for cloud applications is discussed in [7]. Their focus is on enabling the cloud provider to support scalability at the application layer and the work deals with server, network, and platform scalability. Examples of applications that can benefit from such cloud-level enhancements are not discussed. Our work is such an example.

6. CONCLUSION AND FUTURE WORK

In this paper, we studied the dynamic scalability of a SIP proxy system. We proposed a highly scalable architecture, implemented it, and evaluated part of the system. Evaluation was highly restricted due to the use of a public cloud platform. We leave large-scale evaluation of the system on an unrestricted cloud platform as future work.

There are many interesing challenges left unsolved. The LSM can be smarter since it already polls all components in the two tiers. It could proactively manipulate the distribution tables of the load balancer so that the traffic does not go to an overloading SIP proxy. The same logic applies to the DNS server. Instead of relying on round-robin DNS, LSM could potentially manipulate the SRV records to control traffic going to the load balancers.

Also, load balancing strategies for dynamically scaling systems can be different from static systems.

Lastly, high availability is another area left for future investigation.

7. ACKNOWLEDGEMENTS

The authors would like to thank Jan Janak for guidance on SIP Express Router development, Dr. Arata Koike for valuable comments and insights throughout the project, and the anonymous reviewers for their valuable comments and encouragement. We are also thankful to Amazon for the Education Research Grant. This project is funded by NTT. This material is based upon work supported by the National Science Foundation under Grant No. 0751094.

8. REFERENCES

[1] *Secure Copy*, 2011 (accessed June, 2011). `http://www.openbsd.org/cgi-bin/man.cgi?query=scp&sektion=1`.

[2] *Amazon EC2*, 2011 (accessed March, 2011). `http://aws.amazon.com`.

[3] *The Apache Cassandra Project*, 2011 (accessed March, 2011). `http://cassandra.apache.org/`.

[4] *The SIP Router Project*, 2011 (accessed March, 2011). `http://sip-router.org/`.

[5] *Sipp*, 2011 (accessed March, 2011). `http://sipp.sourceforge.net/`.

[6] K. Singh and H. Schulzrinne. Failover, load sharing and server architecture in sip telephony. *Comput. Commun.*, 30:927–942, March 2007.

[7] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41:45–52.