Leaky Bucket Model for Autonomic Control of Distributed, Collaborative Systems

Bogdan Solomon, Dan Ionescu,
Cristian Gadea, Stejarel Veres
NCCT Lab
University of Ottawa
Ottawa, Ontario, Canada
Emails: bsolomon, ionescu, cgadea, steju@ncct.uottawa.ca

Marin Litoiu
York University
Toronto, Ontario, Canada
Email: mlitoiu@yorku.ca

***Abstract*—In recent years cloud computing has taken off and has become an underlying technology in many online applications and websites, whether using a private or public cloud. At the same time with the increase in globalization, online collaborative tools have become very important in order to allow different businesses to communicate with each other, or even different parts of the same company to communicate more easily and share data. Previous work introduced an architecture for a geographically distributed, cloud based collaboration application. While the architecture allowed the scaling up and down of the number of cloud instances being used to host media servers, no control scheme for such cloud scaling was provided. In this paper a model and control scheme for cloud scaling for a collaboration application is provided. The model is based on the leaky bucket model commonly used in network control, while the control system is a self-organizing system. A testbed for the architecture is also presented and used in order to gather performance data from the collaborative application.**

I. Introduction

Cloud computing has proven itself as a technology which has revolutionized the deployment of applications, allowing properly designed applications to scale from serving a small number of people, to hundreds of thousands and even millions while maintaining a small cost when only a small amount of clients are using the system. At the same time cloud computing has decreased the cost of deploying new applications by making use of economies of scale and allowing large companies like Amazon, Microsoft, RackSpace, Google and others to provide hardware and software services which can be used to develop applications which scale automatically.

At the same time, collaboration tools have become a very important part of not only private life but also of companies making business, be it by allowing communication with suppliers or partners from around the world, or simply allowing different offices of the same company from different countries to collaborate. Most of the time, such tools allow some form of video/audio conferencing as well as the sharing of documents, images, videos and even live sharing of the system's local desktop. In [6] such a tool was presented. The collaboration tool in [6] is using a geographically distributed server architecture, in which multiple datacenters, each running a cloud, host instances of the media server for the collaboration application. Clients connect to the closest or best datacenter, determined based on some metric like latency. Clients which connect to different servers or even different datacenters can still communicate and collaborate with each other transparently of the underlying architecture. The architecture uses a Group Membership Service (GMS) system for each cloud which allows servers to dynamically join/leave the group on startup/shutdown such that each of the various clouds can scale up and down. Messages are routed between servers in the same group and between groups such that a proper collaboration state is maintained at all times.

This paper will extend the previous work by adding an autonomic control loop on top of the collaborative application. The autonomic control loop will self-optimize the cloud deployment by scaling the number of servers and virtual machines used based on the demand of each cloud, as well as on the total demand for the application. The control system will be a self-organizing system where each of the servers/virtual machines will run its own control loop and the autonomic control will be achieved by the collaboration between the server control loops, similar to what was described in [7]. The control loop itself will be based on the leaky bucket model similar to that of [2] and [3].

The rest of the paper will be organized as follows: Section II presents the model used for the autonomic control of the system. Section III introduces a test bed used to test the collaborative tool as well as the autonomic control system. Section IV presents some performance data from the test bed. Finally, section V reflects on the contributions of this paper and proposes topics for future research.

II. Autonomic Control Model

In any control system, a very important part is the model of the system. Based on the model created, a system which updates the model using observed data can be chosen and finally a control law can be also built. This section presents the background theory used when building the model and control law for the distributed collaboration application presented in [6].

The autonomic model introduced in this paper is composed of two parts:

978-1-4673-6400-3/13/$31.00 ©2013 IEEE

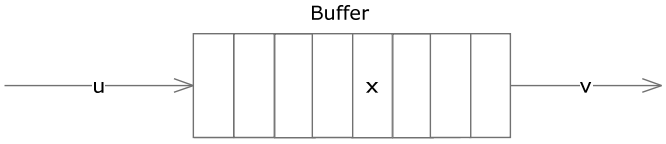

Fig. 1. Single-Server Queueing System

- The leaky bucket model - this is the model which is executed at each server's control loop.
- The self-organizing model - this is the model of how the server control loops are combined to create a control system for the entire cloud.

A. Leaky Bucket Model

The leaky bucket model is commonly used in network congestion control and is based on a fluid flow model like the one in [2]. Due to the nature of the application under control this model has to be modified and the changes to the model will be presented later in this section. In the fluid flow model, a router or other network equipment is modeled as a single-server queueing system with constant service rate, as in Figure 1, where $u(t)$ is the rate of packets arriving at the queue, $x(t)$ is the state of the queue represented by the number of packets in the queue and $v(t)$ is the rate at which packets are processed and released from the queue. This view of the system results in a fluid flow model as follows:

$$\dot{x} = -v + u \tag{1}$$

If the queue operates as a FIFO queue, then the model can be expressed as follows:

$$\dot{x} = -r(x) + u \tag{2}$$

where $r(x)$ is the processing rate of the queue and can be expressed mathematically as the ratio between the state or load of the queue (x) and the residence time of packets ($\theta(x)$) as in equation 3.

$$r(x) = \frac{x}{\theta(x)} \tag{3}$$

As such, the steady state load of the queue depends on the residence time of packets and the residence time of packets could be, for example, linear as proposed in [2]:

$$\theta(x) = \frac{a + x}{\mu} \tag{4}$$

where $a > 0$, $\mu > 0$.

The leaky bucket model is created on top of the fluid flow model, such that the output of the queue is sped up when large bursts of packets appear. This is done by adding a "token bucket" to the queue, which is filled at a constant rate. Whenever a packet is removed from the queue, a packet is also removed from the token bucket. At the same time, the service rate of the queue is controlled by the amount of tokens in the token bucket. The model of the system looks like Figure 2.

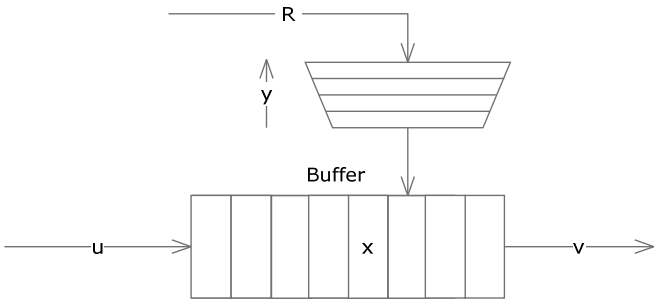

Fig. 2. Token Leaky Queueing System

The token bucket is filled at a constant rate $R > 0$ and $R < \mu$ and the service rate is modulated by y - the fill amount of the token bucket, such that $v = \mu$ when the token bucket contains tokens and $v = R$ when the token bucket is almost empty.

While the above model can be used for control of a queue in networking equipment, a number of modifications have to be made to apply it to the collaboration application described in [6]. The collaboration application provides a number of services - text chat, synchronization of views and audio/video streaming. In order to provide good user experience the system should provide low latencies for the users. Among the various services, the most demanding, and the one which requires the strictest latencies is the audio/video streaming service. View synchronization is done via small messages and requires few server resources. Similarly, compared to audio/video streaming, text chat requires few server resources. Furthermore text messages can be delayed for short periods without perceived loss of user experience. As such, this paper will focus on using the audio/video streaming service as a proxy of how loaded the server is, in order to decide when to add/remove servers from the cluster.

When a user starts streaming audio/video data to the server, the server will take the stream and broadcast it to all other users in the same collaboration session as the streaming user. This can be modeled similar to the fluid flow model, as the server will receive packets containing the streaming data and broadcast them to all the required clients. Since the application server does not track the number of packets received in the various queues, a proxy for the number of packets has to be used. For this, the number of clients streaming to the server as well as the number of streams going out from the server can be used. While any audio/video stream will show burstiness depending on changes in captured video data - still images will use less bandwidth and packets than video data containing movement - and audio data - silence will use less packets, possibly even no packets, than people talking - averaged over an entire server's streams the burstiness will disappear. Because of this we can express the number of packets received by the server as a function as follows:

$$u = nis * pi \tag{5}$$

where the packet rate u is defined as the number of incoming

streams received by the server (nis) multiplied by a constant (pi) which defines the number of packets per stream per time unit. Due to the fact that the server must broadcast the packets to multiple clients, the rate at which packets are being processed is not constant, like in the case of a router for example. As such, the service rate of the server is not constant, but changes periodically dependent on the number of stream receivers. One stream could be broadcast to five receiving users, while another stream is only broadcast to one receiving user and the service rate of packets from the first stream is larger than that of packets from the second stream. Similarly to audio/video data burstiness this disparity is averaged across the entire server, such that the residence time can be expressed as follows:

$$\theta(x) = \frac{a + x}{nos * pp} \tag{6}$$

where the service rate is the number of outbound streams (nos) multiplied by a constant which defines the time a packet takes to be processed by the server (pp).

As such the full continuous time model can be expressed as follows, similar to the work in [2]:

$$\begin{aligned} \dot{x} &= -\frac{nos * pp * x}{1 + x}\frac{y}{\epsilon + y} + nis * pi \\ \dot{y} &= \begin{cases} -\frac{nos * pp * x}{1 + x}\frac{y}{\epsilon + y} + R & \text{if } 0 \leq y \leq \sigma \\ 0 & \text{if } y = \sigma \end{cases} \end{aligned} \tag{7}$$

where σ is the size of the token bucket, which starts full.

B. Self-Organizing Model

The previous subsection presented the model applied to a single server in order to autonomically manage the audio/video stream queue of the server similar to the control of a router's packet queue. In the case of a router, the control of the queue's state is done by simply dropping some packets when they arrive such that the queue does not get filled too quickly and the processing rate is maintained as desired. In the case of the collaboration application the goal is to control the streaming latency by adding more servers to the cloud's pool of servers whenever servers get overloaded. This means that when the control model reaches a point where in the case of a router it would start dropping packets, for the collaboration application it requires the addition of servers to the active server pool. Conversely, when the token bucket approaches full capacity again, servers are removed from the live cloud pool. The addition/removal of servers is achieved by adding self-organizing capabilities to the cloud, similar to the work in [7].

Self-Organizing systems are systems which reach a desired state without the use of any central authority or plan. In order to reach a self-organizing state for the collaboration cloud the server control loops exchange data with each other as well as with the admission control system for the cloud in order to determine if a new server is required or if servers should be stopped. The self-organizing control is achieved in four steps:

1) In the first step, servers which reach a state where they are overloaded based on their leaky bucket control loop ask the admission control to stop redirecting new clients to them. The remaining servers which still receive requests keep track of how many servers receive client connections.
2) In the second step, the admission control decides to add new servers to the cloud if the pool of accepting servers is too small and the accepting servers' token bucket is close to being empty.
3) In the third step, servers which are not accepting new connections and whose token bucket is approaching full state ask the admission control to restart redirecting connections to them.
4) In the fourth step, if a certain number of servers in the cloud have for a period of time their token buckets full, the admission control removes servers from the cloud.

The above four steps ensure first of all that no single server will be overloaded by receiving too many requests compared to its peers. Second of all, by using the state of the token bucket to decide when to add/remove servers to/from the cloud it can be ensured that the number of servers in the cloud does not fluctuate sporadically and exhibit oscillations.

The communication between self-organizing components can be achieved by using a Group Membership Service (GMS) system to broadcast messages among the control peers in order to reach the control decisions. The control messages will be either accept new clients/reject new clients messages sent by the servers to instruct the admission control system of their ability or state messages informing the admission control of the modeled state of the token bucket.

III. Cloud Test Bed

In order to gather data on how the collaboration servers behave, as well as to test the control system a test bed was developed. All servers are currently located in the same location on the same LAN and VLANs are used in order to separate servers into different logical networks. This is done in order to be able to simulate multiple datacenters (clouds) and be able to simulate network load on the connections between datacenters. In the future, a second test bed will be added in a second location with a similar structure but different server capabilities, thus testing how the autonomic control developed in this paper manages a heterogeneous cloud. At the same time, this second test bed will act as a public cloud, while the current test bed will act as a private cloud. Each of the hardware servers in the cluster run OpenStack [4]. On top of OpenStack, each server runs Ubuntu Linux and the collaborative server. The collaborative server is used for enabling communication between clients and is implemented using the Red5 Media Server [5].

Figure 3 shows the physical topology of the infrastructure, which will be used to simulate various deployment scenarios and run tests on how the collaborative system behaves. The test bed uses five servers connected via a switch to one of four routers with a fifth router providing outside internet

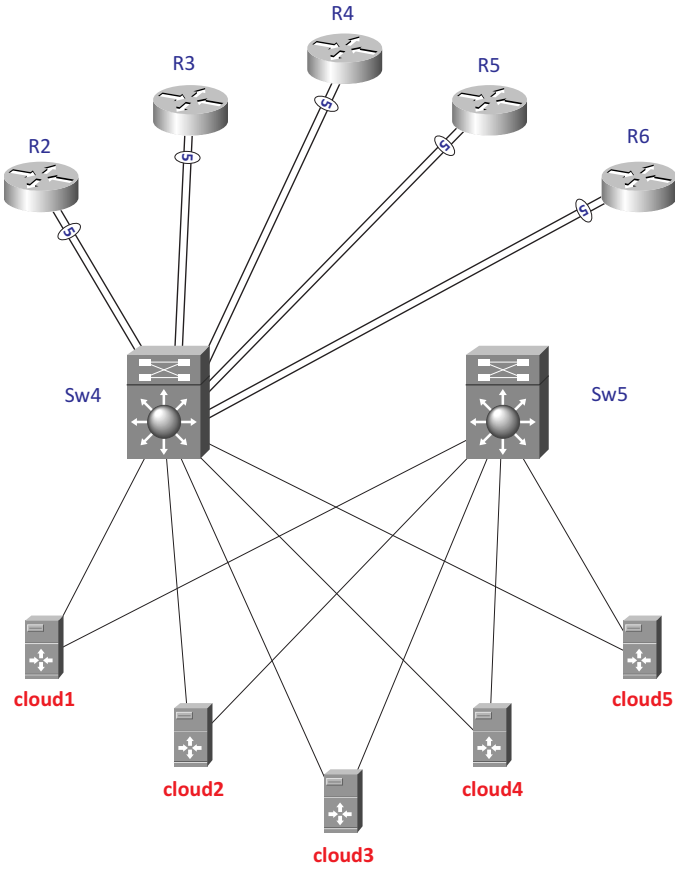

Fig. 3. Physical Topology

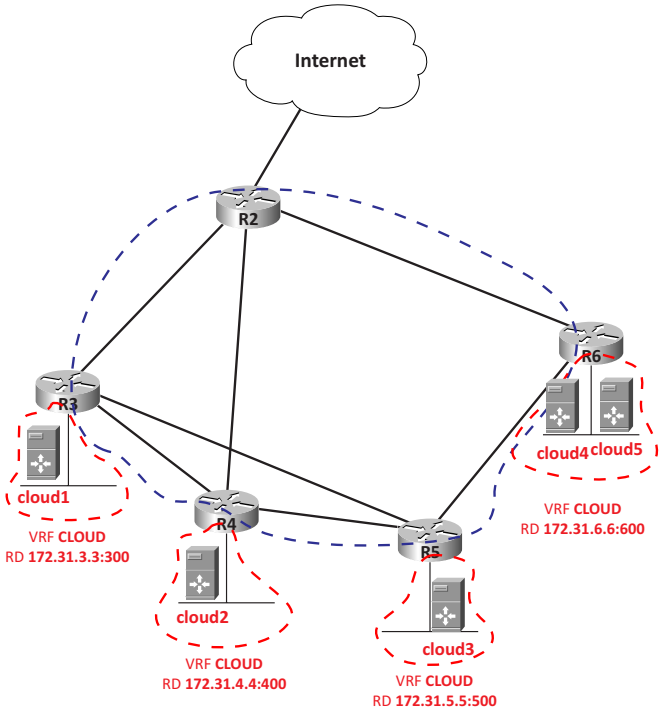

Fig. 4. Logical Topology Routing

connection. Figure 4 displays how routing will be done within the network and the various VLANs used to create the separate clouds. The server names are cloud1 through cloud5, with cloud4 and cloud5 being in the same VLAN, while cloud1, cloud2 and cloud3 are each in their own VLAN. Cloud1 also acts as the cloud controller running all the various services necessary for a cloud like virtual image storage, network management, and cloud computing fabric controller. Each of the virtual machines was given 2GB RAM, 1 VCPU and 20GB hard drive storage.

A separate machine not shown in the diagrams is responsible for simulating client requests. In order to test audio/video streaming a prerecorded webcam video is streamed whenever the client simulator decides to start streaming. The stream is a 125x125 video stream at 25 frames per second. The client simulator is written in Java and can simulate various client distributions by varying the amount of clients, the number of clients in every session, the number of clients streaming in each session and the time delay between messages being sent in a session. The simulator initially creates a number of sessions and a number of clients in each session. Each client is created with a given time to live. Periodically, the session calculates how many clients should be streaming in the session at that point in time. If more clients are required to stream than are currently streaming, the session simulator instructs a number of clients to start streaming also. If less clients are required to stream than are currently streaming, the session simulator instructs a number of clients to stop streaming. If there is no change in the number of clients needed to stream, then no change is made in which clients are streaming. Whenever a client reaches its time to live, the client is put to sleep and given a time after which it should wake up and reactivate. When a client reactivates it joins again the same session it was a member of, before going to sleep. The amount of time clients are awake and sleep is randomized thus generating various session sizes over time.

IV. Results and Parameter Identification

The previously described testbed was used to test the behavior of the cluster of servers in order to determine how the system behaves under various loads, as well as to determine the various parameters of the model like number of packets per stream, and time it takes to process a packet under various loads. In order to measure the data, the server was modified to export certain statistics on request: number of clients connected, number of incoming and outgoing streams, number of proxied streams, average latencies, average bandwidth up and down. On top of these measurements, ntop [1] was installed on the Linux virtual machines and used to obtain extra information like average packet sizes, network load as well as used to graph some of the data.

In order to determine the behavior under various loads, the inputs of the tests were varied as well as the size of the cloud was varied. The following data was varied for the tests:

- The number of servers in the cloud was varied, while holding the number of clients constant at 20, the number of sessions constant at 1 and the number of streaming clients varied. These tests measure how the inter-server stream proxies impact the server's performance.
- The number of clients was varied while holding the size of the cloud constant at 2 as well as the number of sessions constant at 1 and the number of streaming clients varied. These tests measure how the number of clients impacts the server's performance.
- The number of clients was held constant at 20 and the size of the cloud was held constant at 2 while the number of sessions was varied between 1 and 4. These tests measure how varying session sizes impact the server's performance.

For these tests, each test was run for one hour, with sampling done approximately every thirty seconds. Client connections have active/sleeping periods of approximately 500 seconds, with a randomly generated Poisson distribution around the 500 second average.

A. Tests

1) 1 server, 20 users, 1 user streaming: This test is used in order to provide a base in terms of the performance characteristics of the application. Figure 5 shows the bandwidth usage as seen by the operating system via ntop. Note that between 15:40 and 15:50 there was a change in the test being run. As such, this figure is useful for this test up to 15:45. The repeating pattern which is seen approximately every 10 minutes, is due to the fact that the prerecorded video is approximately 10 minutes long, and when it ends, a new stream with the same video is created. From this figure we can see that 20 total streams (1 incoming and 19 outgoing) use approximately 6-8Mbps, which means bandwidth usage of approximately 300-400Kbps for each stream. Looking at the data obtained by measuring directly from the media server, in figure 6, we can see the bandwidth reported by the media server for the user publishing the data stream. Again, the periodic drops to 0 are due to the prerecorded video ending. This figure shows that the bandwidth reported by the server is between 300 and 350 Kbps. Figure 7 shows the bandwidth from the server to the 19 clients receiving data. This figure shows the bandwidth between 5Mbps and 6Mbps for the 19 streams, thus approximately 300 Kbps per stream. This tests show that the measured data at the server is similar to that at the OS level.

Figure 8 shows the distribution of packet sizes for this test at the OS level. While this includes packets for other messages, like DNS queries, DHCP request/replies, etc. the vast majority of traffic (over 99%) is due to the media server. This data can be used to determine the average packet size, which in turn can lead to the number of packets per stream and the variable pi in Equation 5.

2) 1 server, 20 users, 2 users streaming: The results for having 2 users streaming instead of just 1 were very similar to the results of only 1 user streaming. The bandwidth used by

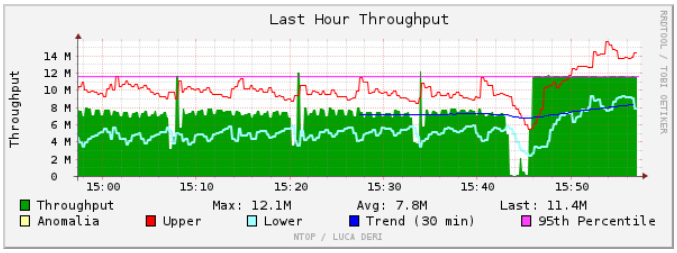

Fig. 5. 1 Server Bandwidth Usage - Operating System

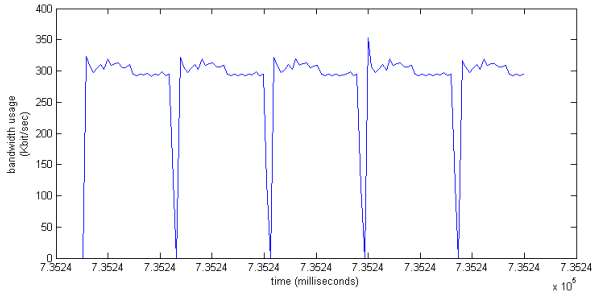

Fig. 6. 1 Stream Bandwidth In - Server

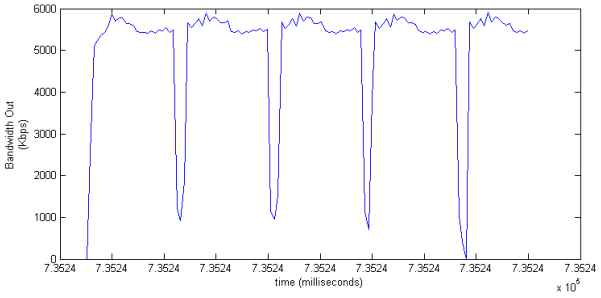

Fig. 7. 1 Stream Bandwidth Out - Server

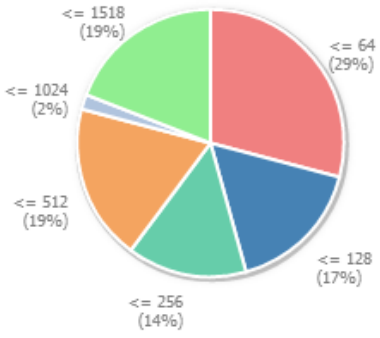

Fig. 8. 1 Server Packet Sizes - Operating System

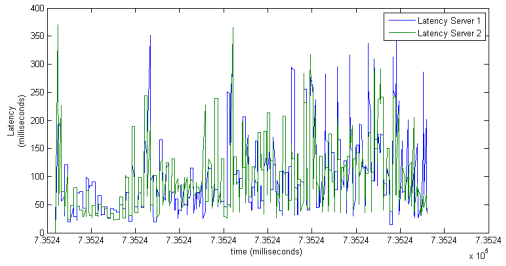

Fig. 9. Connection Latency - 8 Streams

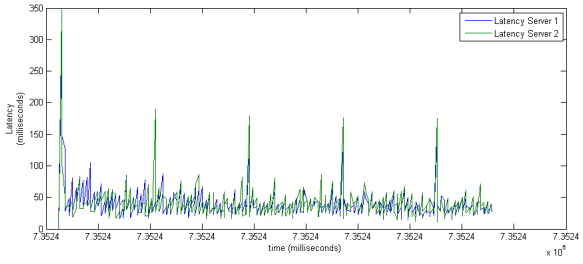

Fig. 10. Connection Latency - 2 Streams

streams received by the server was approximately 600Kbps, with the bandwidth used by the streams broadcast by the server being approximately 9Mbps. It is interesting to note that the bandwidth sent by the server is slightly lower than what would be expected (by doubling the bandwidth for one stream the bandwidth would be expected to be in the 10Mbps to 12 Mbps). This could be due to the media server's re-encoding of the videos or due to imperfections in data sampling. The packet size distribution was similar to that for the previous test,

3) 2 servers, 20 users, 2 users streaming: The results for 20 users spread across 2 servers (10 on each server) where all users are in the same session and 2 users are streaming are a combination of those for tests IV-A1 and IV-A2. The bandwidth received by each server was approximately 600Kbps - 300Kbps streamed directly from client to server, and 300Kbps proxied from one server to another. Bandwidth used for the streams sent by the server was approximately 6Mbps for each server - 1 stream being received by 10 clients and 1 stream received by 9 clients. Packet size distribution was again similar to that of the previous tests.

4) 2 servers, 20 users, 4 sessions, 2 users streaming: With 4 sessions and 2 users streaming, bandwidth received by the server is in the 600 Kbps range, as expected. Bandwidth used for streaming to clients is approximately 1 Mbps for one server and 1.5 Mbps for the second server. Figures 9 and 10 show the connection latencies for tests IV-A4 and IV-A5. From these two figures it is clear that an increase in the number of streams received and sent, leads to an increase in the latency provided to the client. Packet size distribution was again similar to that of the previous tests.

5) 2 servers, 20 users, 4 sessions, 2 users streaming per session: With 4 sessions and 2 users streaming per session, bandwidth received by the server is in the 2.2 - 2.4 Mbps range, with spikes when streams are started. This is fully expected as all tests show approximately 300 Kbps per incoming stream. Bandwidth used for streaming to clients is approximately 4.5 - 5 Mbps for each server. With 4 sessions, each session will have 5 users, spread across both servers - so some sessions will have 3 users server1 and 2 on server2, while some sessions will be the opposite way. With 2 streams received and broadcast to the other clients, this results in the sessions with 2 users having between 2 and 4 streams going out and the session with 3 users having between 4 and 6 streams going out. This will lead to an average of 16 streams going out of each server, and normally being less than that for the bandwidth used in test IV-A1. Packet size distribution was again similar to that of the previous tests.

B. Model Parameters

Using the data obtained from the tests an average value for pi can be determined. Taking packet distribution boundaries (64, 128, 256, etc.) from Figure 8 as the sizes of packets, the average packet size and thus pi is equal to 482 bytes. The other model parameter pp can be approximated by measuring the average latency of the connections.

V. Conclusion

This paper introduced a novel approach for the autonomic control of a collaborative, cloud based application, which uses principles from network control and combines them with a self-organizing mechanism in order to ensure the autonomic scaling of the cloud in view of changing demand. The paper also introduced a test bed for the cloud and autonomic control system and presented performance results obtained from the cloud, which were used to determine parameters of the model.

Future work will focus on providing an implementation of the autonomic system described in this paper as well as using various load simulations in order to determine the behavior of the autonomic system.

References

[1] L. Deri. ntop. ntop. [Accessed: January 2013]. [Online]. Available: http://www.ntop.org/

[2] V. Guffens and G. Bastin, "Optimal adaptive feedback control of a network buffer," in *American Control Conference, 2005. Proceedings of the 2005*, vol. 3, June 2005, pp. 1835–1840.

[3] V. Guffens, G. Bastin, and H. Mounier, "Using token leaky buckets for congestion feedback control in packet switched networks with guaranteed boundedness of buffer queues," in *Proceedings of European Control Conference (ECC)*, 2003.

[4] OpenStack, "OpenStack Cloud Software," OpenStack, [Accessed: January 2013]. [Online]. Available: http://www.openstack.org/

[5] Red5, "Red5 Media Server," Red5, [Accessed: January 2013]. [Online]. Available: http://www.red5.org/

[6] B. Solomon, D. Ionescu, C. Gadea, S. Veres, M. Litoiu, and J. Ng, "Distributed clouds for collaborative applications," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, may 2012, pp. 218 –225.

[7] B. Solomon, D. Ionescu, M. Litoiu, and G. Iszlai, "Self-organizing autonomic computing systems," in *Logistics and Industrial Informatics (LINDI), 2011 3rd IEEE International Symposium on*, aug. 2011, pp. 99 –104.