Self-Optimizing Autonomic Control of Geographically Distributed Collaboration Applications

Bogdan Solomon
NCCT Lab, University of Ottawa
161 Louis Pasteur
Ottawa, Ontario, Canada
bsolomon@ncct.uottawa.ca

Dan Ionescu
NCCT Lab, University of Ottawa
161 Louis Pasteur
Ottawa, Ontario, Canada
dan@ncct.uottawa.ca

Cristian Gadea
NCCT Lab, University of Ottawa
161 Louis Pasteur
Ottawa, Ontario, Canada
cris@ncct.uottawa.ca

Stejarel Veres
NCCT Lab, University of Ottawa
161 Louis Pasteur
Ottawa, Ontario, Canada
sveres@ncct.uottawa.ca

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

ABSTRACT

In the past few years, cloud computing has become an integral technology both for the day to day running of corporations, as well as in everyday life as more services are offered which use a backend cloud. At the same time online collaboration tools are becoming more important as both businesses and individuals need to share information and collaborate with other entities. Previous work has presented an architecture for a collaboration online application which allows users in different locations to share videos, images and documents while at the same time video chatting. The application's servers are deployed in a cloud environment which can scale up and down based on demand. Furthermore, the design allows the application to be deployed on multiple clouds which are deployed in different geographic locations. Previous work however did not introduce how the application's up and down scaling is to be achieved. In this paper the autonomic system which manages the self-optimizing function of the cloud is presented. The autonomic system itself is a self-organizing system with a control model based on the leaky-bucket theory often used in network congestion control. A testbed for the collaboration application is used in order to gather performance metrics for the model.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.2.8 [**Software Engineering**]: Metrics

General Terms

Theory, Design, Measurement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CAC '13, August 05 - 09 2013, Miami, FL, USA
Copyright 2013 ACM 978-1-4503-2172-3/13/08 ...$15.00.

1. INTRODUCTION

Cloud computing has become an integral technology as more and more services make use of the capability to both use hardware resources as a service and to scale a system from a small number of users to millions of users easily. Companies can now make use of outside resources and simply rent the hardware as needed from cloud providers like Amazon, Rackspace, and Microsoft. In the past, scaling an application after release would take weeks as new servers would need to be purchased, configured, tested and finally set to production. With the advent of cloud computing, a new image of a server can be created and be ready to be used in seconds or minutes.

At the same time, the world has become more connected with companies having offices all over the world and with people wishing to communicate with people in far away countries. Due to these reasons, collaboration tools have become more important allowing people to communicate not only by text but also audio/video chat while at the same time share documents, images, videos and collaborate on them. Previous work in [7] presented such a web based collaboration tool. The collaboration tool is cloud based making use of a geographically distributed server architecture, in which multiple clouds at different locations host instances of the media server for the collaboration application. Clients connect to one of the available clouds and can communicate via the collaboration application with clients connected to any other server in any cloud location. Within a cloud, servers use a Group Membership Service (GMS) to communicate with each other such that servers can be dynamically created/destroyed and the servers would join/leave the group as needed. A second level of communication is used through the addition of gateways which enable the communication between clouds. Through the use of these two levels of communication the system can keep the proper state across all the servers, no matter the location of the server.

The geographically distributed cloud based collaboration system requires an autonomic system which would allow the clouds to scale dynamically based on demand. Such a control system has to be able to control the number of servers in each cloud based not only on the local state of each cloud but also on the global state of the entire group of clouds. Such a control system is better achieved by using principles from self-organizing systems which exhibit complex behaviour through the communication between simple components. Each of the servers has its own local control loop and the cloud autonomic self-optimizing behaviour is achieved from the communication between the control loops. Furthermore, through the communication between clouds a global self-optimizing policy can be achieved.

The control loop of each of the servers is based on the leaky bucket model frequently used in network congestion control as presented in [2] and [3]. While the work in [2] is used as a starting point, the model is modified in order to be better applied to the architecture and behaviour of the media server used for the collaboration application.

The remainder of the paper is organized as follows. Section 2 introduces the collaboration application which is controlled by the autonomic system. Section 3 presents the leaky bucket model used for the autonomic control of a server. Section 4 introduces the self-organizing model through which the autonomic computing behaviour is achieved, as well as the controller design. Section 5 introduces a test bed used to test the collaboration tool as well as the autonomic control system. Section 6 presents some performance data from the test bed. Finally, Section 7 reflects on the contributions of this paper and proposes topics for future research.

2. GEOGRAPHICALLY DISTRIBUTED COLLABORATION APPLICATION

In order to better understand the control system developed, the architecture of the controlled system is briefly presented in this section. The entire architecture is presented in [6] and [7]. The collaboration application is a client-server application in which clients connect to a server and upon successful connection see a list of online contacts. Users can then invite online contacts to join them in a collaboration sessions - users can only be in one session at one point in time. Once in the collaboration session, users can communicate via audio, video and text chat. At the same time, the application has a shared section of the display window in which the content loaded is synchronized across all users in the same session. The content of this synchronized section can be a video, an image, a document or a game.

This application was modified in order to allow the server to be deployed in a cloud. One of the most important requirements for the cloud deployment was that the cloud nature of the system be hidden from the end user. As such, end users can view the system as a single, very large server. This design requirement led to the following changes to the single server application.

2.1 Server-To-Server Communication

Since the cloud sizes are not static in order to scale up and down based on demand, a very important ability for the servers is to discover new servers when servers come online and for servers to broadcast when they leave the cluster. In order to achieve this goal, the cluster architecture uses a Group Membership Service (GMS) [1] to create a group which the servers can join and leave. The servers then use the group created by the GMS system to communicate with each other. Using a GMS for such communication instead of an approach like broadcasting the messages in the network ensures that multiple clusters can be co-located in the same network and not interfere with each other's inter-server communication. In order to deal with group members crashing, members periodically send a refresh message to update the status of their group membership. If a long enough period passes without a refresh from a group member, that member is considered dead and removed from the group. If a group member attempts to join a group with a name that does not exist, then a group with that name is created.

2.2 Cloud-To-Cloud Communication Architecture

While the in-cloud server-to-server communication uses lists composed from the cloud's servers, updated via GMS status messages, in order to know where the server peers are located inside the cloud, such an approach would be prohibitive for out-of-cloud communications. At the same time, servers inside the same cloud can know the host names and IP addresses of their peers since the servers are on the same network. In a geographically distributed cloud architecture, servers from one cloud cannot know the host names or IP addresses of servers in other clouds. Even if the servers know these values, the values cannot be used since the servers can be on separate local area networks. Because of this, a tradeoff has to be made between decreasing the number of messages sent between clouds and the global system state stored at each cloud. To achieve cloud-to-cloud communication, the architecture uses a gateway component placed in each cloud. The gateway itself resides on a public IP address. This approach is similar to the one used in [9] which uses brokers to pass data between collaboration clients. The gateway itself is a peer in the GMS group created for the cloud, but instead of processing the messages received like the servers do, the gateway will simply broadcast the necessary messages received to the other clouds. In the case of client-to-client messages, if the destination client is connected to the local cloud - the gateway also stores a list of clients connected locally - then the message is not sent to other clouds. The gateway is also responsible for receiving messages from other gateways, and broadcasting the messages within the cloud.

Note that more reliability can be added to the cloud communication system by using multiple gateways per cloud. Instead of having one gateway for all outward communications, each cloud can have one gateway for communications with each of the other clouds, thus for N clouds, each of the clouds would have N-1 gateways. This approach decreases the stress put on each of the gateways, but requires more public IPs to be used for each of the clouds.

3. SINGLE SERVER LEAKY BUCKET MODEL

In the development of any control system a very important decision, and the decision upon which the correct execution of the control system rests is the decision of what model to use. Based on the model chosen for the controlled system, the control law and measured inputs can be selected. In this section, the background theory and changes to the model chosen are presented.

As previously mentioned, the control of the system is based on the leaky bucket model, which is usually used in order to manage the queues of network routers and switches in order to prevent overloading and is itself based on a fluid

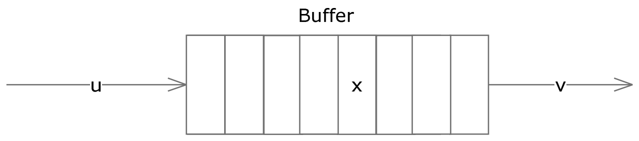

Figure 1: Single-Server Queuing System

flow model [2]. In a fluid flow model, a router or other network equipment is modeled as a single-server queuing 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 $a > 0$, $\mu > 0$. Assuming a constant rate of incoming packets of λ, the steady state load $\bar{x}$ becomes:

$$\bar{x} = \frac{\lambda}{\mu - \lambda} \tag{5}$$

The leaky bucket model is created on top of this fluid flow model, in order to ensure that the network equipment is not overwhelmed when large bursts of packets appear. The control is performed by dropping packets once a certain condition is met. The leaky bucket model is created by adding a “token bucket” to the queue. The token bucket is filled at a constant rate and 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. 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 the control of a queue in networking equipment, a number of modifications have to be made to apply it to the collaboration application previously presented. The collaboration application

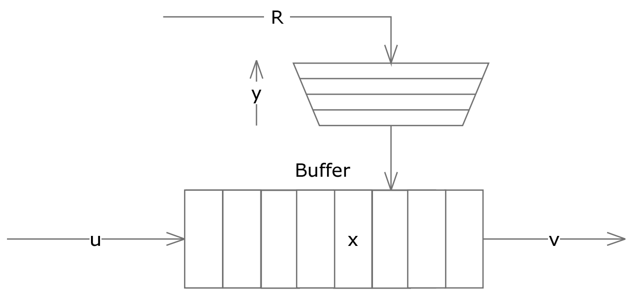

Figure 2: Token Leaky Queuing System

provides a number of capabilities which are under Quality of Service (QoS) considerations: text chat, synchronized view and audio/video chat. In order to provide good QoS the system should provide low latencies for the users. Among the various capabilities, the most demanding, and the one which requires the strictest latencies is the real-time audio/video chat capability. The synchronized view is achieved via small messages and requires few server resources. Similarly, compared to audio/video chat, text chat requires few server resources and text messages can also be delayed for short periods without perceived loss of user experience. As such, this paper will focus on using the audio/video streaming capability as a proxy of how loaded the server is, in order to autonomically manage the load of the servers in the cloud.

When a user starts sending audio/video data to the server for the real-time audio/video chat, the server will take the stream and rebroadcast it to all other users who are in the same collaboration session. This behaviour can be modeled similar to the previous fluid flow model, as the server will receive packets containing the streaming data and rebroadcast them to 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 purpose, either the bandwidth used or the number of clients streaming to the server and 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 depending on settings, than people talking - averaged over an entire server's streams this burstiness will disappear. Because of this we can express the number of packets received by the server as a function as follows:

$$u = nis * pi \tag{6}$$

or

$$u = bw_{in}/p_{size} \tag{7}$$

where the packet rate u is defined either as the number of incoming streams received by the server (nis) multiplied by a constant (pi) which defines the average number of packets per stream per time unit, or as the bandwidth received by the server (bw_{in}) divided by a constant (p_{size}) which defines the average size of a packet. Due to the fact that the server must broadcast the packets to multiple clients, the rate at which packets are being processed can not be considered to be constant, like in the case of a router. 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 how the model deals with audio/video data burstiness, this disparity is assumed to disappear when averaged across the entire server, such that the residence time can be expressed as follows:

$$\theta(x) = \frac{a + x}{nos * pp} \tag{8}$$

or

$$\theta(x) = \frac{a + x}{bw_{out}/p_{size} * pp_{inout}} \tag{9}$$

where the service rate is the number of outbound streams (nos) multiplied by a constant which defines how much processing each stream requires (pp) or the service rate is the outgoing bandwidth (bw_{out}) divided by a constant which defines the average size of a packet (p_{size}) and multiplied by a constant defining how packets in and out are related in terms of processing (pp_{inout}).

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

$$\begin{aligned} \dot{x} &= -\frac{nos * pp * x}{1 + x}\frac{y}{\epsilon + y} + nis * pi \qquad (10) \\ \dot{y} &= \begin{cases} -\frac{nos * pp * x}{1 + x}\frac{y}{\epsilon + y} + R & \text{if } 0 \le y \le \sigma \\ 0 & \text{if } y = \sigma \end{cases} \end{aligned}$$

or

$$\begin{aligned} \dot{x} &= -\frac{bw_{out}/p_{size} * pp_{inout} * x}{1 + x}\frac{y}{\epsilon + y} + bw_{in}/p_{size} \\ & \qquad (11) \\ \dot{y} &= \begin{cases} -\frac{bw_{out}/p_{size} * pp_{inout} * x}{1 + x}\frac{y}{\epsilon + y} + R & \text{if } 0 \le y \le \sigma \\ 0 & \text{if } y = \sigma \end{cases} \end{aligned}$$

where σ is the size of the token bucket, which starts full. In this model $y/(\epsilon+y)$ is a modulation function with $0 < \epsilon \ll 1$.

The above model can now be used in order to manage the audio/video streaming queue of a single server by relating the number of packets coming in, to the state of the queue and the number of packets coming out.

4. SELF-ORGANIZING MODEL

For clarity, in this section server refers to the actual Red5 servers being controlled, server controller refers to the self-organizing controller which controls a single Red5 server, load balancing subsystem refers to the a component inside a cloud which balances client requests between the various active servers in the cloud, cloud control subsystem refers to a component inside a cloud which decides when to add/remove servers and cloud controller refers to the actual cloud mechanism which adds/removes servers.

The previous section introduced the control model applied to a single server's audio/video queue in order to autonomically manage the state of the queue similar to the control of a router's packet queue in order to offer better QoS to the end users via lowered latencies. In the case of a router, the control of the queue's performance is achieved 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 described in this paper, the autonomic goal is to manage the audio/video latency provided by the servers by adding more servers to the cloud's pool of active 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 observed due to the fact that the token bucket is close to empty, in the case of the collaboration application it adds servers to the active server pool. Conversely, when the token bucket approaches full capacity, servers are removed from the active cloud pool. The addition/removal of servers is achieved by adding self-organizing capabilities to the cloud, similar to the work in [8].

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 cloud control subsystem in order to determine if a new server is required or if servers should be stopped. The self-organizing control is achieved in four control decisions:

1. The first decision is done by the servers which reach a state where they are overloaded based on their leaky bucket model. These servers remove themselves from the pool of servers accepting new client connections.

2. The second decision is taken by the cloud control subsystem if the pool of servers accepting new client connections becomes too small and the remaining servers' token buckets are showing that the servers are soon to be overloaded. This implies that the servers share information regarding the state of the token bucket themselves. In this case, the cloud control subsystem adds new servers to the active pool of servers.

3. The third decision is done by any servers which is not accepting new connections and whose token bucket are showing that the servers are not overloaded anymore, bu getting close to being full again. These servers add themselves back to the pool of servers receiving client connections.

4. The fourth decision is taken by the cloud control subsystem if a certain number of servers in the cloud have token buckets showing that the servers are underloaded for a certain period of time. In this case, the cloud control subsystem chooses some servers and stops them, freeing the underlying cloud resources used.

The above four decisions 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, similar to how the servers exchange messages. The control messages are either accept new clients/reject new clients messages sent by the servers to instruct the load balancing subsystem of their ability or state messages informing the cloud control subsystem of the modeled state of the token bucket.

4.1 Controller Design

The previously described equations, 10 and 11 represent the state of the audio/video queue and the state of the leaky token bucket as a function of the number of streams and bandwidth used respectively. In order to develop a controller, the equations have to be modified to a form which can be interpreted by a computer. Also, the data which will be measured from the underlying controlled system has to be determined. This section presents the steps which are taken by the controller in order to apply the model previously described. The controllers described in this section are the self-organizing controller applied to a single server, as well as the cloud control subsystem which adds/removes servers to/from the cloud. The server controller executes the following control iteration every 30 seconds.

1. Initialize at start the token bucket as being full. The size of the token bucket which best represents the underlying system is determined via performance measurements of the system.

2. Retrieve measured data from the managed server. This data includes the number of incoming streams, number of outgoing streams, bandwidth received, bandwidth sent as well as latency and CPU usage. The data received from the server represents in the case of stream numbers the current number of streams, in the case of bandwidth the average bandwidth for the last 30 seconds across all server connections, in the case of latency the latencies calculated for clients in the last 30 seconds. Latency measurements are obtained by pinging the clients and computing the average across all clients, and CPU usage is measured by retrieving the system's CPU usage.

3. Compute a new state for the leaky token bucket by adding R tokens and removing a number of tokens as defined by the equations 10 or 11 based on the previous state of the queue, the previous state of the token bucket and either the number of outgoing streams or sent bandwidth.

4. Compute a new state for the queue by adding packets equivalent to either the incoming number of streams or received bandwidth and removing packets based on the previous state of the queue, the previous state of the token bucket and either the number of outgoing streams or sent bandwidth, similar to the calculations for the new state of the token bucket.

5. Determine the state of the token bucket by comparing it to predefined thresholds. If the token bucket is above the upper threshold, increment a counter for how many consecutive time periods have been above the threshold. If it is bellow the lower threshold, increment a counter for how many consecutive time periods have been bellow the threshold. If it is between the two thresholds, reset both counters to 0.

6. If the upper counter has passed a threshold of how many successive time periods can be above the threshold, request that the server no longer accept new clients.

7. If the lower counter has passed a threshold of how many successive time periods can be bellow the threshold and the server is not accepting new clients, request that the server start accepting new clients.

It should be noted that the number of successive periods bellow the lower threshold before starting to receive new clients should be lower than the number of successive periods above the upper threshold. This is due to the fact that a client which is not receiving new clients and which is being underloaded is very unlikely to become overloaded. On the other hand, a server which is overloaded can become very easily underloaded as clients leave or stop streaming.

An extra control loop resides on top of the server control loops - although by being a member in the GMS group it could be seen as being a peer in the self-organizing system - and performs two control actions:

1. Add/remove servers to/from the cloud by making requests to the cloud controller. Upon reception of messages from the servers regarding the state of their token buckets the cloud control loop computes how many servers are overloaded and based on a predefined threshold adds new servers. Similarly, if the number of underloaded servers passes a predefined threshold, servers are removed from the active pool.

2. Talk to other cloud controllers to determine which clouds new clients should be redirected too. This control sequence is also done in a self-organizing manner:

    (a) Cloud controllers communicate with each other and exchange information regarding number of total servers, number of active servers and number of overloaded/underloaded servers.

    (b) Upon start-up, clients ping all the clouds' load balancing subsystems to determine which cloud to connect to.

    (c) Each of the load balancing subsystems reply, but with a delay dependent on the cloud controllers inter-communication. As such, the cloud controllers set a delay value in each of their respective load balancers, based on the load of each cloud.

Through the combination of the delay set by the cloud controller and ping time, clients connect to a cloud which offers both good network connectivity and good load.

5. COLLABORATION APPLICATION TEST BED

In order to gather data on how the collaboration servers behave, as well as to test the control system a small test bed was developed in which various loads were applied to a cloud of media servers and the necessary data was measured from the servers. All the 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 data centers (clouds) and be able to simulate network load on the connections between data centers. 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 collaboration server. The collaboration 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 was used to simulate various deployment scenarios and run tests on how the collaboration system behaves. The test bed uses five servers connected via a switch to one of four routers with a fifth router providing outside internet connection. The second switch connects the servers to a management network. Figure 4 displays how routing is 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. The network connections are 100Mbps with some of the router connections being 10Mbps. While such connection speeds would be too low for a data center, it is fine for these tests as the low speeds can be used to simulate overloaded internet connections with low throughput.

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 which was used for testing was a 64x64 video stream at 25 frames per second with a bit rate of 180Kbps. 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 allowing for the generation of various session sizes over time.

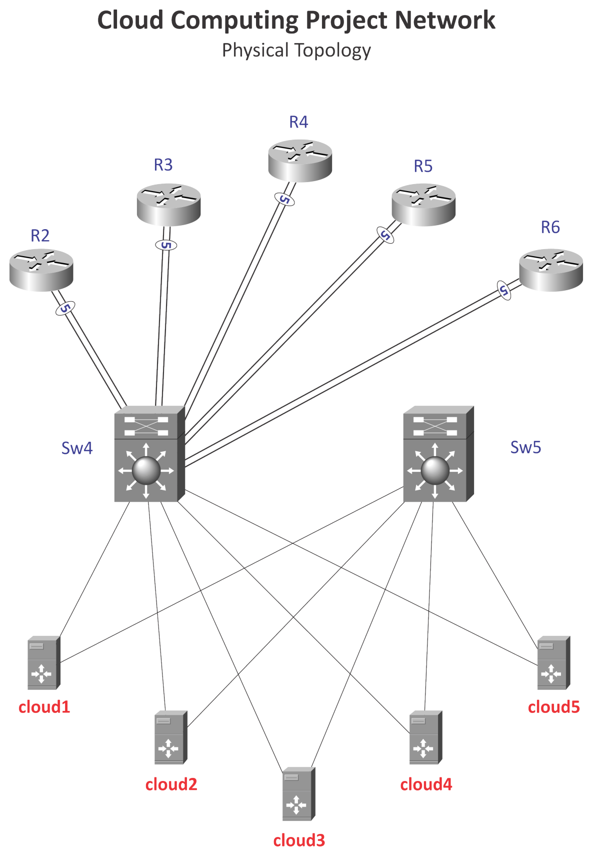

Figure 3: Physical Topology

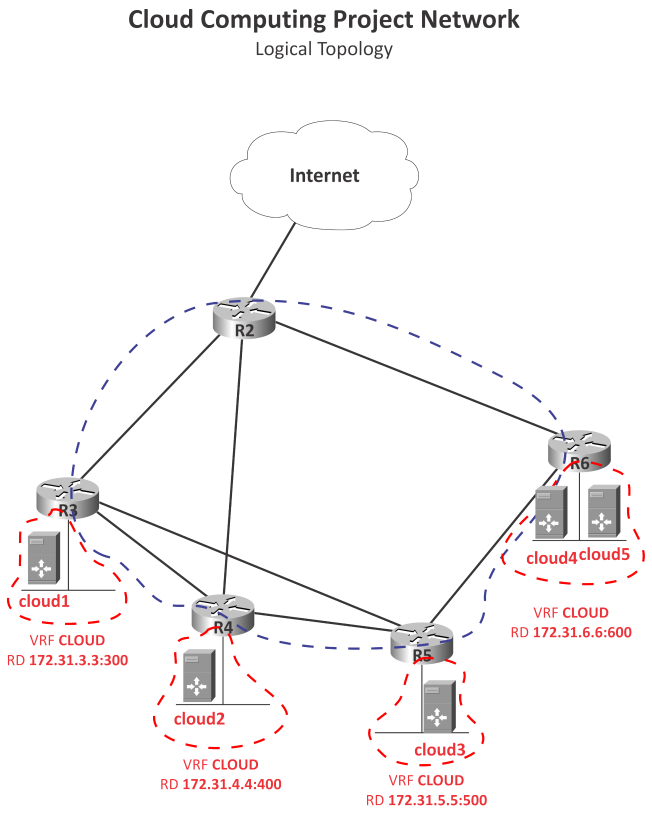

Figure 4: Logical Topology Routing

Table 1: Median and Mean CPU, Latencies for 1 Server, 1 Session, 1 Stream

Number of Users	Median Lat. (ms)	Mean Lat. (ms)	Median CPU (%)	Mean CPU (%)	Out Streams
5	10.40	13.15	6.86	6.16	4
10	11.60	13.20	14.51	12.41	9
15	12.03	24.08	21.49	18.31	14
20	14.90	30.43	27.56	23.57	19
25	16.48	48.05	33.41	28.50	24
30	20.02	65.88	38.15	32.79	29
35	34.80	109.41	42.36	36.64	34
40	59.75	135.60	44.21	38.62	39

6. RESULTS

There are three variables which can be varied in order to determine how they affect the measured data obtained from the collaboration servers. For these tests one variable was modified while all others were kept constant. The four variables are the following:

1. Number of collaboration sessions in each server, varied from one to three.

2. Number of clients in each session, varied in increments of five from five to fourty. In some cases, the increases in clients stopped before reaching the maximum due to the 10Mbps network link becoming saturated.

3. Number of clients streaming per session, varied from one to three.

The data is measured from all the servers every 30 seconds, and is composed of: bandwidth received, bandwidth sent, latency and CPU usage. Since the tests are run at different moments in time, the timescale is normalized to the period from when each test was started.

The expectations are that as the number of servers increases for a given number of clients, sessions and streams the bandwidth and latency will decrease proportionally. Similarly, as the number of collaboration sessions increases, with all other variables kept constant bandwidth and latency will decrease as the 1:N proportion, where N is how many clients receive the stream decreases. When the number of clients in a session increases bandwidth and latency should increase in a reverse fashion. Finally, when the number of streaming clients increases bandwidth and latency should increase similarly. For these tables Latency was abbreviated to Lat.

6.1 1 Server, 1 Session, 1 Stream Per Session

This test is used in order to provide a base in terms of the performance characteristics of the application. The number of clients was varied from 5 to 40, where the 10Mbps network link was nearly saturated, in increments of 5 users. Table 1 shows the median and average latencies and CPU usage as the number of users increases. This test shows that latency and CPU usage are not directly correlated. Between 30, 35 and 40 users latencies start increasing substantially, however CPU usage continues to grow at a steady rate and even slows down between the 35 and 40 user tests.

Table 2: Median and Mean CPU, Latencies for 1 Server, 1 Session, 2 Streams

Number of Users	Median Lat. (ms)	Mean Lat. (ms)	Median CPU (%)	Mean CPU (%)	Out Streams
5	10.20	11.92	13.56	11.33	8
10	12.79	21.64	22.95	19.12	18
15	20.03	64.12	31.87	27.54	28
20	33.95	95.57	40.15	31.35	38
25	277.84	274.80	35.31	32.28	48

Table 3: Median and Mean CPU, Latencies for 1 Server, 1 Session, 3 Streams

Number of Users	Median Lat. (ms)	Mean Lat. (ms)	Median CPU (%)	Mean CPU (%)	Out Streams
5	10.40	19.77	19.86	16.44	12
10	14.55	35.68	34.44	28.76	27
15	66.53	120.20	38.82	33.46	42

6.2 1 Server, 1 Session, 2 Stream Per Session

The results in table 2 show the mean and median latencies and CPU usage for 2 audio/video streams in one session. The number of clients varied from 5 to 25, in increments of 5, due to the fact that at 25 the 10Mbps link is saturated. For each of the various numbers of clients the number of outgoing streams is double that of 1 stream per session. Looking at latency values it can be noticed that for a similar number of outgoing streams, latency mean and median are very close, while CPU usage varies by quite a bit. For example, the 30 users test case for 1 stream has latency mean: 65.88 ms and median: 20.02 ms, while the 15 users test case for 2 streams has latency mean: 64.12 ms and median: 20.03 ms. For the same test comparison CPU usage is quite different between the two tests.

6.3 1 Server, 1 Session, 3 Stream Per Session

The results for 3 streams in one session from table 3 show that the number of outgoing streams is not the only factor in observed latencies. 27 outgoing streams, with only 10 users in a session (3 streams, each going to 9 users) exhibits much lower latency values - mean: 35.68 ms and median: 14.55 than 28 outgoing streams, with 15 users in a session (2 streams, each going to 14 users) - mean: 64.12 ms and median: 20.03 ms.

6.4 1 Server, 2 Sessions, 1 Stream Per Session

The test case with 2 sessions and 1 stream per session, proves that latency is a function of the number of users and number of outgoing streams. 20 users per session (resulting in 40 total users) with 2 incoming streams and 38 outgoing streams shows higher latency values - mean: 126.36 ms and median: 63.51 ms than 38 outgoing streams, with 20 total users (2 streams, each going to 19 users) - mean: 95.57 ms and median: 33.95 ms.

6.5 Packet Sizes

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 5 as the sizes of packets, the

Table 4: Median and Mean CPU, Latencies for 1 Server, 2 Sessions, 1 Stream Per Session

Number of Users	Median Lat. (ms)	Mean Lat. (ms)	Median CPU (%)	Mean CPU (%)	Out Streams
5	10.95	13.04	13.79	11.61	8
10	12.75	31.38	23.25	19.82	18
15	20.12	60.83	32.39	28.00	28
20	63.51	126.36	41.10	35.99	38
25	365.65	341.45	35.19	32.16	48

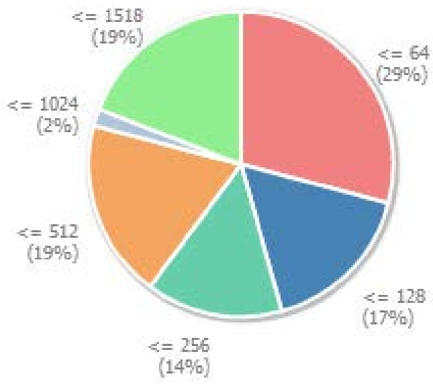

Figure 5: Server Packet Sizes - Operating System

average packet size and thus *pi* can be determined to be equal to 482 bytes. The other model parameter *pp* can be approximated from the previous latency measurements as a function of number of users, number of streams and outgoing streams.

7. CONCLUSIONS

This paper introduced a novel approach for the autonomic control of a cloud based application responsible for providing user collaboration tools, which uses principles from network control and combines them with self-organizing mechanisms in order to ensure the self-optimizing function for the cloud due to changing demand. The paper also presented performance results obtained from the cloud, which were used to determine some parameters of the model.

Future work will focus on providing a full implementation of the autonomic system described in this paper as well as using various load simulations in order to better determine the model parameters and also test the behavior of the autonomic system and determine how fast the system adapts to changes and how well the cloud resources are used. The system will also be tested with higher speed networks and will simulate varying network latencies.

8. REFERENCES

[1] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36:37–53, December 1993.

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

[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. `http://www.openstack.org/`. [Accessed: January 2013].

[5] Red5. Red5 Media Server. `http://www.red5.org/`. [Accessed: January 2013].

[6] B. Solomon, D. Ionescu, C. Gadea, and M. Litoiu. *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*, chapter Geographically Distributed Cloud-Based Collaborative Application. IGI Global, 2013.

[7] 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*, pages 218 –225, may 2012.

[8] 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*, pages 99 –104, aug. 2011.

[9] A. Uyar and G. Fox. Investigating the performance of audio/video service architecture. II. Broker network. *International Symposium on Collaborative Technologies and Systems*, 0:128–135, 2005.