# Autonomic Control of On-the-Cloud Applications for Distributed Collaborative Systems

Bogdan Solomon, Dan Ionescu, Stejarel Veres
NCCT Lab, University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada

Marin Litoiu
York University
4700 Keele Street
Toronto, Ontario, Canada

Gabriel Iszlai
IBM Center
for Advanced Studies
Toronto, Ontario, Canada

Octavian Prostean
"Politehnica" University
of Timisoara
Timisoara, Romania

***Abstract*—On-the-Cloud computing became an establish domain of research which materialized in a variety of *as a Service applications. One of the issues with cloud computing relates to the complexity of the IT operations needed to install and maintain the *as a Service infrastructure. Autonomic Computing Systems have been seen as a solution for providing the "*as a Service" with self management, due to their ability to configure, optimize, heal and protect themselves with little to no human intervention. The Autonomic systems must be able to analyze themselves and their environment -the *as a Service infrastructure- in order to determine how best they can achieve the high-level goals and policies given to them by system managers. Previous work on Autonomic Systems has focused on management of resources which are collocated in the same datacenter. However, new services like Content Delivery Networks require that the servers which provide the service be distributed across different datacenters. At the same time web applications are increasingly used for people to collaborate within the same organization or among organizations. This paper focuses on the issues related to the design and implementation of an Autonomic Computing infrastructure specially built for autonomically managing an *as a Service system by applying known patterns from real-time control to the on-the-cloud deployed collaborative application. The autonomic system architecture for the self-management of on-the-cloud distributed applications is approached through an abstract application model while real-time control patterns are applied to the combination of the self-* parts of the architecture and the on-the-cloud distributed application. The test bed for a practical deployment of such a system together with some tests performed, are provided at the end of this paper.**

## I. Introduction

With the advent of new paradigms for the deployment and efficiency of application oriented software products, the IT departments have become more pervasive and more complex leading very often to unmanageable IT infrastructures. Failures of those infrastructures can have disastrous consequences for the company and sometimes for a region or a country's economy. In a world that evolves as fast as the current one, an IT failure can result in lost sales and even the loss of customers to competitors for a company.

Cloud computing exacerbates these issues an IT infrastructure must face by moving the IT outside the company's premises. Where before each small enterprise would run its own small IT department, now large data centers provide IT services to multiple consumers and enterprises (SaaS, HaaS, IaaS). For the IT providers the ability to maintain the systems' service level agreements and prevent service outages is paramount since long period of failures can open them to large liabilities from their customers. At the same time, cloud computing provides the ability for companies to pay only for the required resources and to scale up or down as more resources are needed or as resources are no longer required. Due to this capability, a solution is needed for cloud computing users in order to intelligently decide when to request more servers and when to release used servers. From the point of view of cloud computing providers, a solution is needed in order to move server loads such that only the required CPU power is used for a certain demand via virtualization. Server virtualization also increases the complexity of managing the servers in data centers, since suddenly one single hardware server can be running tens of virtual machines, each with its own load and processing requirements. Ensuring that the appropriate number of virtual machines are deployed on a hardware platform, such that the hardware is neither under-utilized, nor that the virtual machines starve each other for resources is not a trivial administrative task.

These are the problems that autonomic management systems attempt to solve. Autonomic computing systems are capable of self-managing themselves by self-configuring, self-healing, self-optimizing and self-protecting themselves, together known as self-CHOP. Such a system must be able to analyze itself at runtime, determine its state, determine a desired state and then if necessary attempt to reach the desired state from

the current state. Normally the desired state is a state that maintains the system's Service Level Agreement (SLA). For a self-configuring system for example, this could include finding missing libraries and installing them with no human intervention. A self-healing system would be able to determine errors in execution and recover to a known safe state. A self-optimizing system example would be a cluster of servers that dynamically adds and removes servers at runtime in order to maintain a certain utilization and client response time. Finally, a self-protecting system example would be a server that detects a denial of service (DoS) attack and prevents it by refusing requests from certain Internet Protocol (IP) addresses.

Furthermore, companies have policies which do not allow certain data or mission-critical applications to be moved outside the premises of the company. As such, companies are developing clouds which are part public and part private - called hybrid clouds. These systems must also be managed, and due to the split between private and public clouds the management is further complicated. Some of these systems will require that data from the public cloud be used by applications in the private cloud, and possibly that this data is combined with data stored in the public cloud. In order to optimize the performance of such a system, both the public and private clouds must be optimized separately, as well as the combined use case. The problem becomes even more complicated when it comes to collaborative applications. Such applications, have different SLA requirements then standard web applications. While response time is very important for a normal web application, latency and jitter are critical to any collaborative application which includes video or audio streaming between parties.

In this paper a model driven architecture for the on-the-cloud computing and a real-time architecture for the autonomic computing capable of self-CHOPing the on-the-cloud computing environment based on control loop models will be introduced. The design and implementation of a network based test bed on which these principles were tested are described as well. The issues that must be dealt with, in order to autonomically manage an *as a Service system by applying known patterns to the whole infrastructure are then discussed. The paper considers, as a case study, an application which enables real-time collaboration among its users. The application studied has its resources distributed in multiple datacenters over a production network called NCIT*net 2. The autonomic system architecture for the self-management of the on-the-cloud distributed application which enables the users' collaboration is approached via the application model. The real-time control patterns will be applied to the combination of the self-* part of the architecture and the on-the-cloud distributed application.

The approach proposed in this paper is based on an architecture rooted in real-time system theory and on software engineering patterns and practices, presented in [8] and [9]. The rest of the paper is structured as follows: Section II introduces in depth the use case for this problem. Section III briefly presents the foundation of the proposed architecture. Section IV presents the approach taken in order to self-optimize the application described in section II. Secion V presents some results and finally, section VI presents the conclusions and future work to be done.

## II. Distributed Collaborative Application: The Use Case

As described in the introduction, the use case for a distributed autonomic system is a real-time collaborative application, which has its server nodes distributed in multiple datacenters. Each of these datacenters is created as a cloud computing platform and extra virtual machines can be created in the cloud in order to increase the capacity available. In this section the application which will be optimized is introduced together with some of the requirements for the autonomic control system which will optimize the application.

The collaborative application provides the capability for multiple clients, whom are together in a session, to share the same view of either video, images or presentations synchronously. At the same time, the application provides the capability of video/audio chat as well as text chat between the participants in a session. Multiple exclusive sessions - one user can only be in one session at any point in time - can exist on the server, and the server provides delimitation between the sessions. Each of the three shared views have slightly different use cases in terms of synchronization:

1) Image sharing - this is the simplest of the three. Users can either share a single picture, or a slideshow of pictures. In the case of a single picture the application ensures that all the users in a session can view the picture. In the case of a slideshow, the application ensures similarly that all the users view the same picture at the same time. This means that as the slideshow moves from one picture to the next all users' pictures are updated. At the same time, users can pause or restart the slideshow, which has to be propagated to all the users in the session. Users can also choose to skip a number of pictures in which case all users will skip to the same picture in the slideshow.
2) Video sharing - in the case of video sharing, users can choose the video which they wish to share, in which case all users queue the same video. Users can also choose to skip to a certain point in time in a video, in which case all users in the session will skip to the same point in time in the video. Finally, users can pause/resume/stop the video which is synchronized across all users. Sound controls are not synchronized across all users.
3) Presentation sharing - users can also share presentations or documents created in any document processor application. In the case of both documents and presentations, the application ensures that the all users view the same page/slide and in the case of pages that do not fully fit in the viewer area that all users view the same area of the page. Zoom levels are however, not synchronized across all users.

More on the specifics of the application can be found in [4].

### A. Autonomic Computing Problems

In terms of deployment the collaborative application which is to be optimized is composed of two primary parts:

1) A client part which runs on the user's machine, in the browser as an Adobe Flash [1] application
2) A server part which is responsible for providing the communication links between clients, as well as ensuring that all the clients in a collaborative session are synchronized to the same state. Since the clients are Flash based applications, the server is allowing Real-Time Messaging Protocol (RTMP) [2] connections from the clients.

At its simplest, the system can be composed of a single server and a number of clients. However, such a deployment will obviously not scale as more users connect to the server and will not provide any redundancy. A simple approach to the scaling and redundancy problems would be to create a cluster of servers, which using autonomic computing principles would scale up/down as demand increases/decreases. Due to the fact that the application provides video/audio chat latency becomes extremely important however. As such, a better approach would be to have multiple datacenters spread across different locations, with each datacenter being able to scale up/down based on demand. When a user connects to the system, the user is connected to the datacenter which provides the best latency for the user. This is done as the server implements geolocation mechanisms. In most cases this is the datacenter closest to the collborating users. For a non-collaborative system, such a deployment would be sufficient, however as users join and leave sessions a mechanism is added. This mechanism either enables server-to-server communication or moves users from one server to another in order to have most of the users in sessions located on the same server. Both solutions present extra problems for an autonomic computing solution.

If server-to-server communication is added, the autonomic self-optimization function must take the extra load of this communication in consideration. In previous work [7], a control function for the number of servers was developed based solely on the number of clients connecting to the server. However, if server-to-server communication added, the number of clients connecting is no longer sufficient to describe the state of the system. Consider a system with three servers and three users all in the same session. In such a case the users could be connected in three ways:

1) All users connected to the same server - no server-to-server communication needed, number of clients sufficient to describe the state of the system
2) Two users connected to the same server, third user connected to a different server - two one way server-to-server links have to be created in order to pass messages between the users.

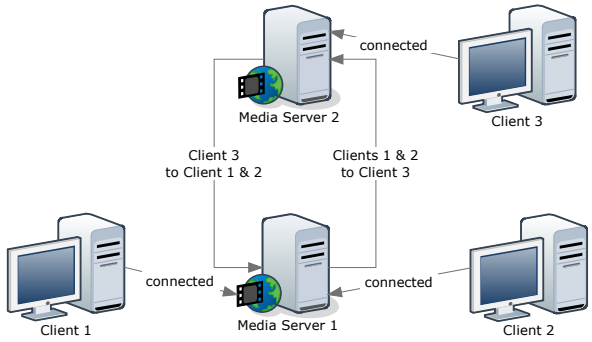


Fig. 1. Server-To-Server Communications

3) Each user connected to a different server - six one way server-to-server links have to be created in order to pass messages between the users.

Figure 1 shows the second use case.

If clients are moved between servers as they join/leave sessions the autonomic computing system must be able to deal with quick shifts in the number of clients connected to a single datacenter. Consider the same example given before, and assume that each of the three users connects to a different server initially. Also assume that each of the servers is at capacity. When the new session is created with the three users in it, the server chosen to host the session suddenly sees a spike in usage, while the other two servers suddenly become idle. However, when the session ends the users are moved back to the original servers, resulting in the two idle servers suddenly hitting capacity, while the server which hosted the session supporting more connections.

The approach of connecting clients to the best datacenter introduces an extra problem - what to do when a new client joins but the datacenter is at capacity. In such a case, the autonomic system brings a new virtual machine up with a new server which joins the existing cluster. While the virtual machine is started the client can either be made to wait, or can be connected to a different datacenter which still has capacity opened. When the virtual machine has started, the client can be moved to the originally chosen data center.

## III. Autonomic Management System as a Service

The distributed autonomic computing system presented in this paper is based on the autonomic architecture presented in [8], where the base common unified architecture for autonomic systems was introduced with the goal of breaking the autonomic system into its base components. Based on this architecture, a repository of components is created, with the components working together to form an autonomic system. Using this repository as a starting point and object composition software patterns, an autonomic system is built by dynamically combining the appropriate low level services provided by the components. A system administrator, either a human user or an automatic system, can select the correct components from the repository and configure the components to work together such that they form a managing system for an external resource. At

the same time, such an architecture allows components from various vendors to interoperate, as the architecture provides standard interfaces that must be exposed by different components. Furthermore, this approach generates a system that is capable of modifying its structure at runtime, by simply replacing a component in the system with another component which exposes similar services. The composability of the system is achieved by enforcing that communications between components are done only via the component interfaces, which represent the service contract between components.

Figure 2 shows the high-level architecture of the autonomic system, based on the eight components described in [8]. Each of the eight components performs only one role, thus providing high cohesion and low coupling. The architecture defines all the eight components, however a control loop does not need to be composed of all them, if it does not make sense logically to use certain services. For example a control loop that uses raw data for its analysis and decision services can use the data directly from the sensors and is not required to use a filtering service, and so on.

The architecture is composed from the following eight components:

1) Sensors - The sensor performs the monitor function in the control loop. Its single goal is to gather raw data from the managed resource. The sensor does not process the data gathered, and only makes it available to the next component in the system. As such the sensor must interface with the underlying resource, retrieve the necessary data and format the data to the required output.
2) Filters - The filter's role is to take data coming from sensors and modify it based on the requirements of other components. The filter communicates with the sensors in order to retrieve data available in the sensors, processes this data based on some internal structure and makes the data available to other components in the control loop.
3) Coordinator - The coordinator is the brain of the autonomic system. Its function is too coordinate the execution of the other components in order to achieve the final goal of the autonomic system. It is responsible for gathering data from the filters and using this data and its own internal logic to decide the order in which to execute the other components.
4) Model - The model acts as the knowledge base of the system. It is used in order to both predict the future state of the system if no changes are made as well as to predict the changes that would be needed to maintain the desired system outputs. A valid and stable model is paramount for an autonomic system. The model must be able to represent the state of the system accurately enough for decisions to be made regarding changes in the system.
5) Estimator - The estimator acts as the predictor of the control loop. Its goal is to use the modeled data and any new measured data in order to estimate the future state of the system. Like the model, the estimator can use various approaches to determine the future state like machine learning methods or control theoretic methods. Since the decision of making modifications to the system is based on estimated data, the estimator must provide stable estimates in the face of changing environments.
6) Decision Maker - The decision maker is the component which generates a change plan, if one is needed. Based on a request coming from the coordinator, which asks for a decision to be made regarding necessary changes in the system, the decision maker uses the model's data, its own internal logic as well as preset knowledge in the form of QoS and SLA levels in order to generate a decision which maintains the managed resources' SLA and QoS.
7) Actuator - The actuator is equivalent the execution part of the control loop. It is responsible for executing a change plan created by the decision maker. This can include scheduling actions for a later date, communicating with various components of the resource and requesting changes, as well as executing workflows. The actuator also ensures that in the case of a failure in the execution, the action is rescheduled for a later time if possible.
8) Adaptor - The adaptor is necessary in order to ensure that the control loop performs correctly. Its responsibility is to adapt the rest of the autonomic system to changes in the managed resource. Unlike the model which changes based on measured data and estimated data, the adaptor makes changes based on structural changes to the managed resource. The adaptor can also be used to make structural changes to the control loop if there are changes in the environment.

On top of this architecture, a framework was built which provides messaging, reliability, and validation mechanisms to the autonomic components. Through the use of this framework, which is developed as an extension to the capabilities provided by the WSDM standard, and of the autonomic repository which was also built, an autonomic system can be created as a combination of services provided by the components. These services can be discovered via the repository, with different services residing on separate machines, and they can be composed to form the autonomic managing system via the cloud.

## IV. Autonomic Management System for Distributed Collaborative Systems

Using the architecture and framework in section III a solution is developed for the self-optimization of the system described in section II. The goal of self-optimization is to optimize the number of virtual machines running in each of the datacenters, such that resources are used efficiently. In such an optimization problem, two conflicting issues are balanced. On one hand, the desire is to use as few virtual machines as possible in order to minimize the energy costs and upkeep for the servers. On the other hand the response time and latency of the servers must be within desired values in order not to impact the application's clients. As such, the problem is one of minimizing the number of servers used, while at the same

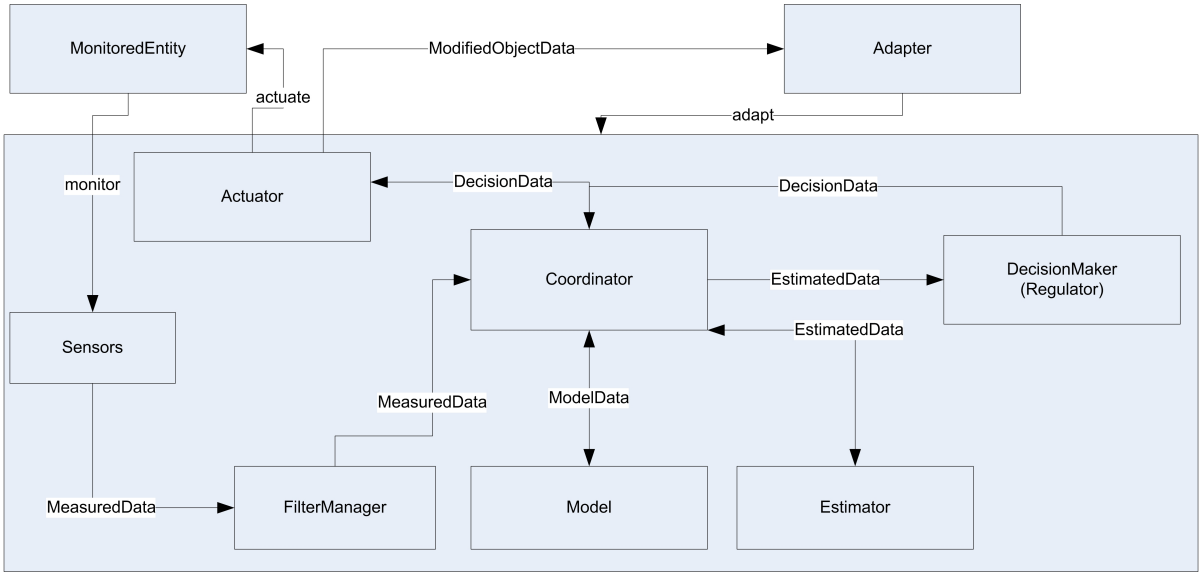


Fig. 2. Autonomic Control Loop

time maintaining the desired response time and latency. One extra problem rises due to the long time it takes to create new servers. While the desire is to minimize the response time and utilization, actions to remove or add servers do not happen unless they are absolutely needed. For example, if removing a server improves the utilization maintaining the same response time or latency, but the improvement in utilization is small no action is taken in order not to disturb the system.

While this is similar to the classical problem of self-optimization, the system previously described requires extra mechanisms to ensure that not only are the datacenters optimized locally, but that they are also optimized globally. In order to achieve this, a two layered system was developed, where two layers of control loops execute in parallel. At the lower level, each datacenter has its own control loop which is responsible for locally optimizing the datacenter. This control loop starts and stops virtual machines as needed based on user demand in order to provide the required latency and response times. However, this is not sufficient to ensure global optimization of all the datacenters. Since each of the low level control loops only monitors the local servers and virtual machines it is impossible to know the usage of other datacenters, which in turn makes it impossible to predict possible spikes in demand due to sessions being created across datacenters. Two possible solutions can be used to ensure that the system is optimized globally.

The first solution is to have the low level control loops communicate among themselves and exchange information regarding each datacenter's state in order to create a global model of the entire system. The advantage of such an approach is that the autonomic system scales itself as more datacenters are added, but at the cost of extra communications - which can lead to slower autonomic decisions - and the cost of replicating the same information across all the low level control loops.

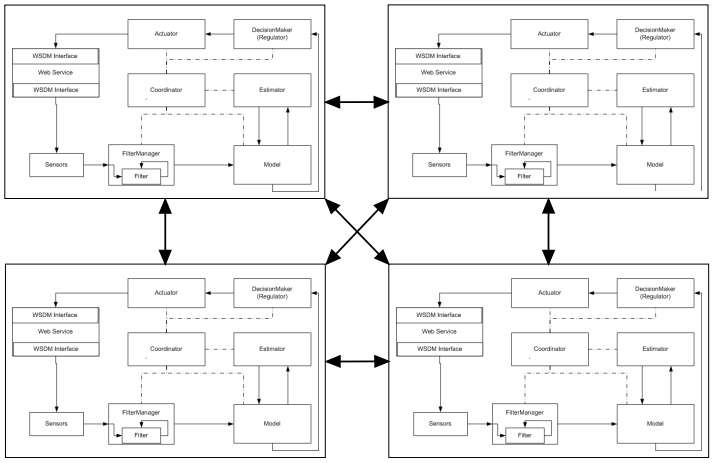

Fig. 3. Control Loop Cross Communication

Figure 3 shows how such a system works in the case of four low level control loops.

The second solution is to have a high level control loop which gathers data from the low level control loops, and whose decisions are passed back to the low level control loops in order to guide the decisions of these loops towards a global optimum. This is similar to the work done in [3] where a high-level controller makes decisions which can not be determined locally at the low levels. The disadvantage of such a system is its ability to scale - as more datacenters are added, the high level control loop has more data to gather and more data to process. At the same time however, the low level control loops can be maintained fairly simple and there is no data replication as the global state is maintained by the high level control loop. Figure 4 shows the hierarchical system, with four low level control loops and a high level control loop. The data and

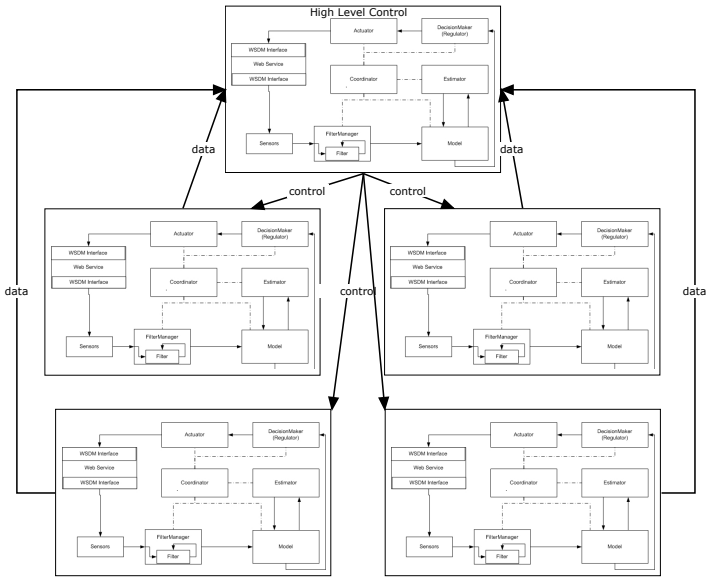


Fig. 4. Hierarchical Control Loops

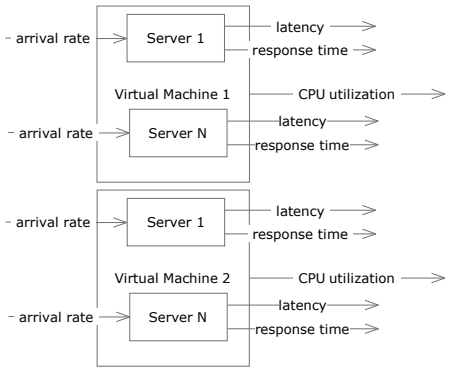


Fig. 5. Low Level Control Loop Model

control sequences from low level to high level and high level to low level do not necessarily follow a request-response pattern. It is possible that the high level control loop gathers multiple data points from the low level ones before needing to make modifications to the low level control loops. Modifications to the low level control loops are done through the use of the adaptors; the high level control loop sends a control message to the adaptor, which in turn modifies the low level control loop.

The autonomic system described in this paper uses the second solution - two-layered control loops in order to self-optimize. The two layers will split the responsibilities in order to reach the global optimum as follows.

### A. Low-level control loop

Each of the datacenters have its own low-level control loop, which attempts to optimize the virtual machines in the datacenter. The control loop gathers data from the virtual machines in the datacenter, as well as from the servers running on top of the virtual machines in order to determine the number of clients currently connected to the servers, the CPU utilization of the servers, as well as the bandwidth used for video/audio streaming in the datacenter. The data is aggregated across the servers by simple average, since it is assumed that all the servers in one datacenter are similar. Based on this gathered data, and on any guidance coming from the high-level control loop, the datacenter autonomic system predicts future usage of the datacenter's resources through the use of a Kalman Filter, similar to the work done in [6]. Once the prediction is done, the autonomic system determines if a breach of SLA will happen due to current and future usages, and resource availability. If such a breach is predicted, the system computes the number of virtual machines which can prevent it and ensures the deployment of the virtual machines. Data gathered regarding usage, clients and number of virtual machines is passed to the high-level control loop. Figure 5 shows the black box model of the low level control loop. Each of the servers has as input function the arrival rate of new users. This in turn acts on the latency and response time of each of the servers and on the global CPU utilization of the Virtual Machine hardware the servers are running on.

### B. High-level control loop

The high-level control loop performs a number of actions in order to ensure that all the datacenters are optimal and that clients experience good performance. First of all, the high-level control loop acts as an access control point. When a client first attempts to open a connection to the application's servers, the connection is done by the high-level control loop. Since the high-level control loop has a view of the entire system - where datacenters are located, how much usage each datacenter has, how many free resources each datacenter has - it is the most appropriate decision making point as to where to redirect the client. The decision of where to redirect the client is done based on two criteria as described before: latency between client and datacenters and usage of each datacenter. If the best datacenter in terms of latency is at capacity, the high-level control loop instructs the control loop in charge of the datacenter to start a new virtual machine and redirects the client to another datacenter while the virtual machine starts. Once the low-level control loop notifies the high-level one that the resource has started and is ready to accept connections, the client is transparently transfered to the new server. A problem can appear at transfer time if the client has already been moved to a different server due to a session which contains other users. In such a situation, the question is what to do with the new server - stop it in order to conserve power or keep it running in case the user leaves the session. Currently, the server is kept running for a short amount of time and if no need for it appears, the high-level control loop allows the low-level control loop to free the resource. The decision of actually

stopping the server is done by the low-level control loop.

Second of all, the high-level control loop attempts to predict global use patterns across all the datacenters. This means that the high-level control loop must be able to predict how users will move from one server to another due to session setup and tear down. Due to server start times, it is not sufficient to create the new servers when the session is created. The servers must be already available to transfer all the users in one session to the same datacenter, and preferably the same server. The high-level control loop use either machine learning [**?**], or statistical rules to try to determine sessions sizes and session locations. These two options can be selected by the system administrator as a function of his/her confidence in such systems.

A simpler solution than having the servers standing by for possible sessions is to create the new servers when the sessions require them and while the servers are created use server to server communication to enable the sessions. Once the servers are started, the sessions and the users are transfered to the appropriate new servers. The advantage of this approach is that the servers are only running when needed and there is no waste of servers running without clients. The disadvantage is that until the new servers are started all the clients of the servers which intercommunicate will suffer a degradation in quality.

Finally, the high-level control loop is responsible for monitoring the execution of the low-level control loops in order to ensure that all the low-level control loops behave as expected and that high-level SLA goals are met. For example, in order to optimize its own behaviour the high-level control loop monitors the time it takes to deploy new servers in each of the datacenters. Based on this monitored value, the high-level control loop attempts to determine if any of the low-level control loops is experiencing problems deploying new servers. A timeout value is set, and if the datacenter has not successfully deployed a new server, then the request is aborted and another datacenter required to deploy a new server. At the same time, the deploy time of the new server is used in order to optimize the location where new servers are deployed. Datacenters with low deployment times are used over datacenters with high deployment times. Figure 6 shows the black box model of the low level control loop. For each of the various clouds arrival of new clients, creation of new sessions and deployment of new servers impact session sizes, deployment times of new servers and number of servers in the cloud.

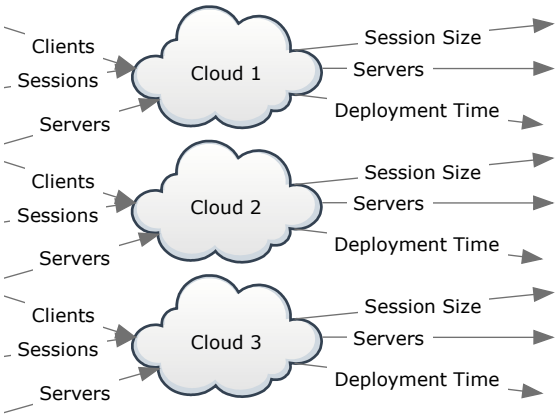


Fig. 6. High Level Control Loop Model

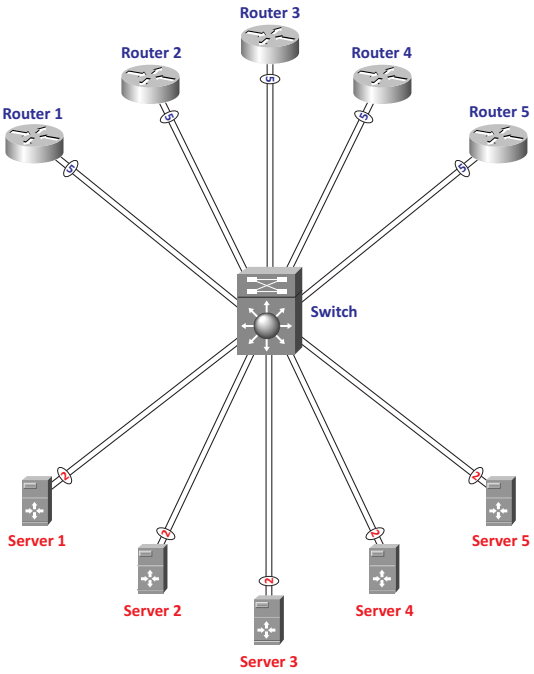


Fig. 7. Physical Topology

## V. Results

In order to develop and test the autonomic system described previously, a test bed was designed and implemented. While all the servers are currently located in the same geographic place, VLANs were created in separately addressable networks in order to separate servers into different on-the-cloud clusters representing different datacenters. Each of the hardware servers in the cluster which supports virtual machines run Eucalyptus [5]. On top of Eucalyptus, each server runs Debian Linux and the open source Red5 server is used for enabling the RTMP communication between clients.

One of the machines in each VLAN is responsible also for executing the low-level control loop for that 'datacenter', while the high-level control loop executes on a machine outside the VLANs. Figure 7 shows the physical topology of the infrastructure which was used to simulate various deployment scenarios and run tests on how the autonomic system behaves. The test bed uses two groupd of five servers connected via a switch to one of five routers in order to obtain the logical deployment shown in figure 8. Finally figure 9 displays how routing is done within the network and the various clouds.

A separate machine is responsible for simulating client requests. In order to test various locations for clients, when making a connection clients will also send a location infor-

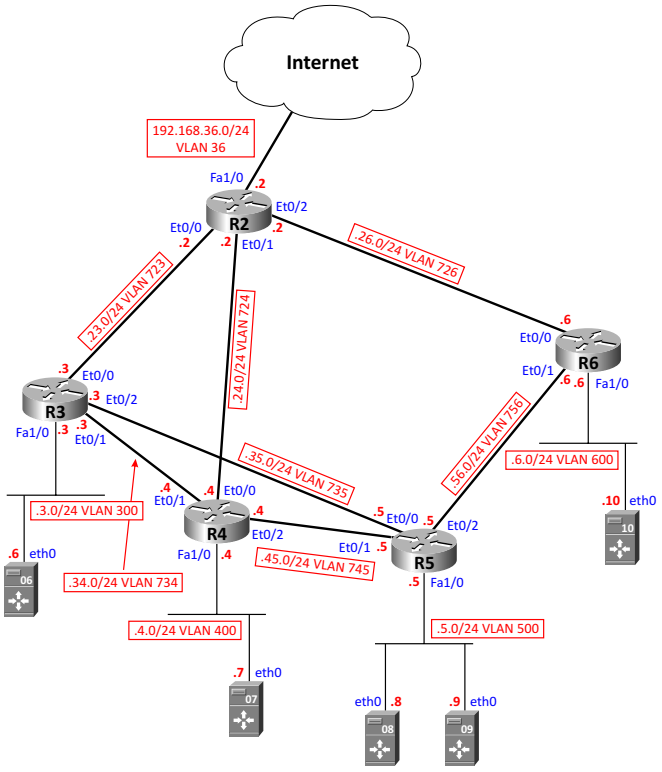


Fig. 8. Logical Topology

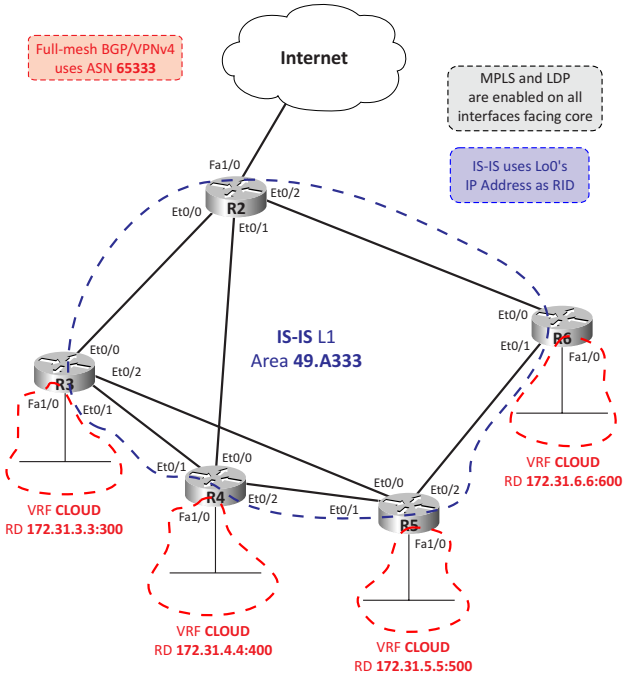


Fig. 9. Logical Topology Routing

mation which will be used to decide which server to use for the client.

## VI. Conclusions

This paper introduced an approach for developing autonomic systems for collaborative distributed systems where the distribution is of the type on-the-cloud. The paper first presented challenges caused by the collaborative and distributed nature of the system under control, and examined different approaches through which those challenges can be overcome. The proposed autonomic computing approach focuses on a two layered control loop where the low-level controllers act on the distributed datacenters and the high-level control loop ensures that the decisions made by the low-level controllers are globally optimal. Furthermore, the high-level control loop can act as an admission control system for the clients' requests.

While the subject approached in this paper is wide, there are other researches taking place right now focussed on a number of issues such as:

- Design, implementation, and performance testing regarding the model for the high-level control loop;
- Design and implementation of various deployment scenarios in order to evaluate and measure the performance of the control loops;
- Design and implementation of appropriate admission control algorithms in the high-level control loop

## VII. Acknowledgments

The authors would like to thank the IBM Centre for Advanced Studies (CAS) for their support of this research and for the technical help.

## References

[1] Adobe. Adobe Flash Player. [Accessed: September 2010].

[2] Adobe. Real-Time Messaging Protocol (RTMP) Specification 1.0. [Accessed: September 2010].

[3] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed & parallel application programming. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –12, May 2009.

[4] R. Dagher, C. Gadea, B. Ionescu, D. Ionescu, and R. Tropper. A SIP based P2P architecture for social networking multimedia. pages 187 –193, oct. 2008.

[5] EucalyptusSystems. Eucalyptus Cloud Computing. [Accessed: September 2010].

[6] E. Gelenbe. Autonomic adaptation in distributed systems and networks preliminary version. 2009.

[7] M. Litoiu, M. Mihaescu, D. Ionescu, and B. Solomon. Scalable adaptive web services. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 47–52, New York, NY, USA, 2008. ACM.

[8] B. Solomon, D. Ionescu, M. Litoiu, and G. Iszlai. Composition of adaptive web services. In *SEAMS '10: Proceedings of the 2010 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 1–10, 2010. To be published.

[9] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu. Towards a real-time reference architecture for autonomic systems. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 1–10, 2007.

[10] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu. Model-driven engineering for autonomic provisioned systems. In *COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1110–1115, Washington, DC, USA, 2008. IEEE Computer Society.