UNIVERSITY OF OTTAWA

Self-Organizing Autonomic Computing Systems for Geographically Distributed Clouds

Bogdan Solomon

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the PhD degree in
The Ottawa-Carleton Institute for Electrical and Computer Engineering

"The important thing is not to stop questioning. Curiosity has its own reason for existing."

Albert Einstein

Abstract

Recently a great deal of research has been undertaken in the area of automating the enterprise IT Infrastructure. For enterprises with a large number of computers the IT Infrastructure represents a considerable amount of the enterprise budget assigned to its operation. Autonomic Computing Systems are systems which were created to correct and optimize the IT infrastructure's own self-functioning by executing corrective operations without any need for human intervention. Applications of autonomic systems range from usage optimization for clusters of servers or virtual machines, to error prevention and error recovery for web applications. In most cases where autonomic computing systems have been developed this was achieved by the addition of external global controllers monitoring the subsystems of the enterprise IT Infrastructure, determining where changes should be made and applying appropriate commands to implement these changes. Self-Organizing systems on the other hand are systems which reach a global desired state without the use of any central authority. At the same time the software industry has seen an explosion in the usage of cloud based services. By moving services in the cloud companies can decrease the IT budget such that they only use the resources which are needed, when they are needed. Furthermore, services deployed in the cloud can be distributed geographically across multiple datacenters such that users obtain a higher quality service by connecting to a closer datacenter. In this thesis proposal, an autonomic system based on a self-organizing architecture is introduced. The system is applied to the self-optimization of a web-based real-time collaborative application running on a geographically distributed cloud. Experiments and simulations with the self-optimizing autonomic computing methodology are given to support the design and the implementation described.

Acknowledgements

I would like to first thank my supervisor, Dr. Dan Ionescu, for the guidance and encouragement that I received during my Master's and PhD work as well as for the oppurtunity of doing this degree. At the same time I would like to thank Dr. El Saddik for starting me, while I was still an undergraduate student, on the path that lead here.

Special thanks also go to all the people in the NCCTLab at University of Ottawa who allowed me to bounce ideas of them, and listened to my endless complaints.

Last but not least, I would like to thank my parents and my brother for always cultivating in me the need to know more.

Contents

List of Figures

List of Tables

Abbreviations

ACM	**A**ssociation for **C**omputing **M**achinery
ARM	**A**pplication **R**esponse **M**easurement
CDN	**C**ontent **D**elivery **N**etwork
DMTF	**D**istributed **M**anagement **T**ask **F**orce
EC2	**E**lastic **C**loud **C**omputing
EKF	**E**xtended **K**alman **F**ilter
EVO	**E**nigmatec **V**irtual **O**rchestrator
FCFS	**F**irst **C**ome **F**irst **S**erved
FEC	**F**orward **E**rror **C**orrection
GMS	**G**roup **M**embership **S**ervice
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
i.i.d.	**i**ndependent and **i**dentically **d**istributed
JEE	**J**ava **E**nterprise **E**dition
JMX	**J**ava **M**anagement e**X**tension
JSON	**J**ava**S**cript **O**bject **N**otation
MAPE	**M**onitor **A**nalyze **P**lan **E**xecute
MDD	**M**odel **D**riven **D**evelopment
N1 SPS	**N1 S**ervice **P**rovisioning **S**ystem
OASIS	**O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards
PIM	**P**latform **I**ndependent **M**odel
PSM	**P**latform **S**precific **M**odel
QNM	**Q**ueueing **N**etwork **M**odel
QoS	**Q**uality **o**f **S**ervice

RPE	**R**ecursive **P**rediction **E**rror
SLA	**S**ervice **L**evel **A**greement
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
TIO	**T**ivoli **I**ntelligent **O**rchestrator
TPM	**T**ivoli **P**rovisioning **M**anagement
WSDM	**W**eb **S**ervice **D**istributed **M**anagement

Dedicated to my parents and my brother

Chapter 1

Introduction

1.1 Motivation and Research Objectives

As IT departments have become more pervasive and more important for each and every company, as well as to our daily lives, they have also become more complex. In today's world people obtain their news and alerts online, shop online as well as communicate with friends and family via social networking sites. The complexity of the IT infrastructure has lead however to a situation where it is nearly impossible for humans to continue to manage and maintain the IT resources in a good state. Failures of the IT infrastructure in a company can have disastrous consequences for the company or even for the economy. In a world that is as fast as the current one an IT failure can result in lost sales, loss of customers to competitors or even payment of damages depending on how critical the infrastructure is.

Cloud computing exacerbates these issues by moving the IT infrastructure outside a 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 cloud 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 underutilized, nor that the virtual machines starve each other for resources is not a trivial administrative task. The issue becomes even more complex when the end user's location is taken into consideration. In order to achieve better response times and latency it is preferable to offer services as close as possible to the end user. Such approaches can be seen in Content Delivery Networks (CDN) [1], which cache web data in datacenters across the world in order to be closer to the end users. A similar approach is taken by Netflix in order to cache the most viewed shows and movies closer to the customers.

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 run time, 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 run time 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.

The above goals of autonomic computing were mapped in the Manifesto [2] on eight key requirements that a system must meet to be considered autonomic. An autonomic computing system must:

1. "Know itself" by knowing its components, status, ultimate capacity, and other interconnected systems

2. Configure and reconfigure itself under various and sometimes unpredictable situations. The configuration must be done automatically by the system

3. Never settle for the status quo, and must always try to optimize the available resources, itself, and the way it works

4. Be able to heal itself in case of malfunctions on some of its parts, and be able to recover its normal state of execution

5. Be able to protect itself by identifying and responding to threats against various types of attacks, while maintaining system integrity and security

6. Be aware of its environment and act according to the environmental needs

7. Function in a heterogeneous and open way, and implement open standards

8. Keep its complexity hidden from the end user

Since the release of the Manifesto a number of research directions and a corresponding number of projects have been developed to look at how to create such intelligent systems which can substitute automatically generated operative commands to human intervention. Various approaches have been taken to reach the desired system self-adaptation. In terms of how the self-management behavior is reached two main architecture approaches have been used. In the first approach a global controller is added to the system, which gathers data from the various components, performs some form of analysis on the data, compares the analysis results with the desired goals and if the goals are breached by the predicted analysis, corrective actions are taken. Such approaches are seen in [3], [4] and [5]. In the first type of approaches a cluster of servers for example, would have one controller which manages the entire cluster's adaptation. The second approach attempts to develop an autonomic system by developing small intelligent subsystems which achieve global self-adaptation through the interactions with other components. Such approaches can be seen in [6] and [7]. In this second approach each server in a cluster would have its own intelligence which achieves local adaptation. The communication between components is then used in order to reach global system adaptation. Such systems can be developed by looking at Self-Organizing systems which are systems that are capable of reaching a desired state without the use of any central authority or plan. Such a system can be seen in Ashby's homeostat [8] which is capable of adapting itself to any perturbation in the system and reach

back a stable state. Self-Organizing Networks have also been developed in recent years for mobile networks.

In this thesis proposal, a Self-Organizing autonomic system which self-manages a cloud of servers is introduced. The system design and results of experiments obtained during system's tests are presented. The servers run a real-time collaborative application which allows the end users to share the same view, as well as to chat via text and video/audio streams. The cloud can be deployed in multiple locations and still allow any two users to communicate with each other while each user connects to the closest datacenter. The self-managing function ensures that all the servers in the same data-center maintain the same response time and CPU usage. At the same time, the self-organizing system can add or remove servers as needed when users connect or disconnect from the service.

The motivation to develop a geographically cloud based collaboration application is related to providing better performance parameters to users by locating the servers closer to the clients of the service. With the servers being placed closer to the clients, latency observed by clients can be lowered in most cases. The application is deployed on top of a public or private cloud such that it can be scaled by adding or removing virtual instances of the application.

The motivation for developing an autonomic computing system to manage the cloud deployments is related to providing intelligent scaling of the cloud resources such that only the required resources are used at a given point in time. With the autonomic system managing the cloud resources, it can be ensured that servers are idle when users do not need them. At the same time when demand increases the autonomic system ensures that more servers are started such that the latency and response time of the system are maintained to desired levels.

1.2 Organization of the Thesis Proposal and Research Focus

The thesis proposal is organized such that it presents the self-organizing autonomic system moving from the design of the autonomic control system and of the geographically distributed application under control to various mechanisms which have to be set up to ensure the self-optimization of the cloud resources usage.

Simulation results as well as results from the system run-time are also shown.The rest of the thesis proposal comprises the followings.

Chapter 2 presents the goals of the self-organizing system and also presents various related work in both autonomic architectures and self-organizing problems.

Chapter 3 presents the models developed for autonomic computing environments used for self-organization of clusters of application servers as well as for the collaborative application described in the thesis proposal.

Chapter 4 introduces the system under control, i.e. the real-time collaborative application, its cloud deployment and its functionality. This chapter also presents the requirements for self-optimization of the application.

Chapter **??** describes the self-organization approach for the resource control of the application server cluster and for the collaborative application.

Chapter **??** presents the test environments used for determining the performance of the architecture proposed, and of the models used.

Chapter **??** describes simulation results of the system as well as results obtained by running the self-organizing system presented on a test bed.

Finally chapter **??** presents conclusions of the benefits provided by the proposed solution. Future developments of the architecture are also described.

The thesis proposal's research focus will be on a number of contributions to the field of autonomic computing and distributed systems, as presented in [9], [10], [11], [12] and , [13].

First of all, the research will focus on a new mathematical model for the Autonomic Computing Environment. This model will be a control theoretic model and will be based on a fluid flow model describing in mathematical equations the arrival and departure of requests from a server's queue. The model will provide better performance for auto-scaling of the system when compared to previous approaches. The developed model will be extended to use a self-organizing controller which is added to the Autonomic Computing Environment. The self-organizing approach used for server scaling in data-centers and cloud systems will allow heterogeneous resources to be part of the same server cluster, while normally a self scaling system will use homogeneous only resources.

Second of all, the research will focus on extending a single server real-time collaborative application into a geographically distributed cloud application. The geographic distributed system will provide better latency to its clients by allowing users to connect to servers closer to their location and which offer better Quality of Service (QoS). The distributed nature of the system will be hidden from the users, such that from the point of view of users everyone is connected to a single server.

A test-bed is used in order to obtain results on the performance of the self-organizing autonomic system while managing the geographically distributed collaborative application.

Chapter 2

Autonomic Systems and Self-Organizing Systems

This chapter presents the known approaches for autonomic computing and self-organizing systems and motivates the need for developing a self-organizing architecture for autonomic systems, including the advantages that such an architecture would provide. The chapter also introduces prior related work in the area of real-time collaborative systems due to the fact that the system under control is such a system.

2.1 Reasons for Self-Organizing Architecture

Since the introduction of autonomic systems by IBM [14], autonomic systems have been developed using the Monitor, Analyze, Plane, Execute (MAPE) loop. Figure 2.1 shows the loop as described by IBM. The resource under control, which can be as simple as a single server or as complex as a cluster of servers, is constantly being monitored to determine changes in its state. This monitoring can be either periodic measurements of certain values in the system, like CPU utilization or response time, or alarm based when certain conditions are breached and an alarm is raised. Once the measured or alarm data is obtained by the monitor function, the data is passed on to the analyze function which uses its internal logic and rules in order to analyze the data and to determine if the system exhibits any problems. If a problem is discovered, the problem information is sent to the plan function whose role is to determine an appropriate response to fix the existing problem or

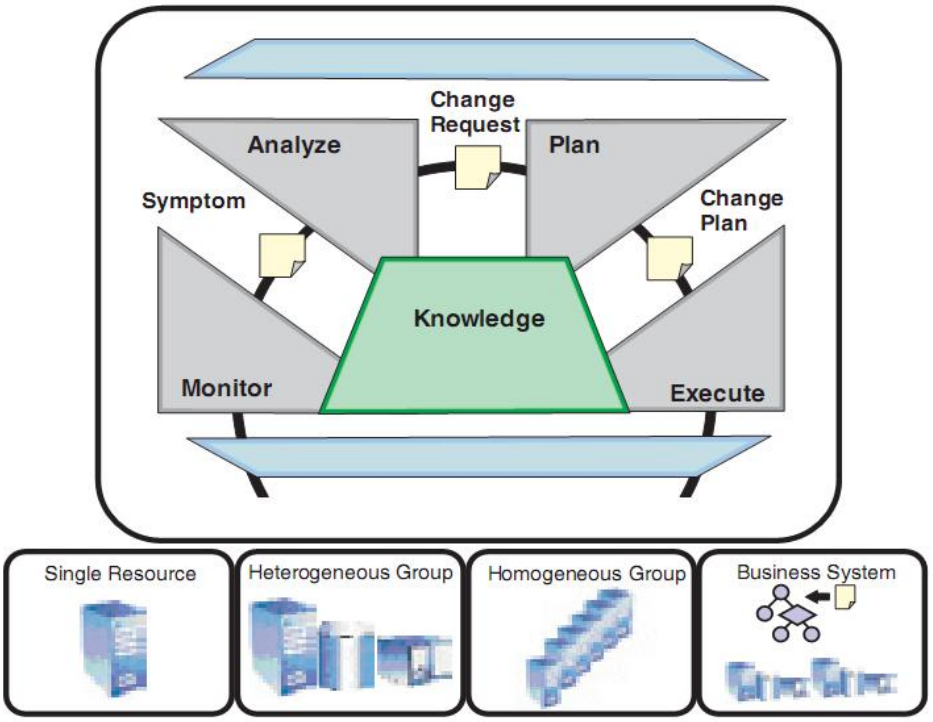

FIGURE 2.1: MAPE Loop

problems. The plan function creates an execution plan which will fix the problem and bring the controlled resource back into compliance. The execution plan is then delivered to the execution function which puts the plan into execution by making modifications to the resource and to the resource's environment.

In order to illustrate the autonomic system approach consider a cloud of servers. The goal of the autonomic system is to maintain a good utilization of the server resources, while at the same time guaranteeing a certain level for the cloud's response time. For such a case, the autonomic control loop would measure all the servers response times and CPU usage. If the response time becomes too high, as determined by the analyze function, the plan function calculates how many new servers to add to the cloud. Once the number of new servers is determined, the execute function is responsible to bring those servers online by starting them and adding them to the cloud. If the CPU usage of the servers in the cloud decreases too much however, the plan function calculates how many servers to remove from the cloud. It is then the responsibility of the execute function to stop the correct servers, remove them from the cloud and free any other resources used by the servers.

Two main approaches have been taken in research in order to build autonomic systems. The first approach considers the entire system under control as a single unit and attempts to add a global control loop which controls the entire system. In the above example, such a system would gather data from all the servers and use some form of mathematical approach like an averaging of the measured data in order to estimate the state of the entire cloud. Based on this average the system would then determine any breach of Service Level Agreement (SLA) and create a plan if a breach exists. This approach has a number of assumptions on how the cloud is formed and how the cloud behaves. First of all, the autonomic control system assumes all the servers in the cloud are exactly the same. If the cloud was heterogeneous, then it would not be possible to average the measured data to obtain a global view of the cloud - unless a more complex procedure is used where the analyze function also knows ahead of the time the relative power of all the servers. Second of all, the system assumes that all the servers not only are the same, but behave the same as well and also that the client requests are balanced appropriately across all the servers. If this was not the case, it would be possible to have for example two types of requests A and B, where A type requests are much more computationally intensive than B and where all the A requests go to Server 1 and B requests to Server 2. Such a case could lead to an undesirable situation where the response time of Server 1 is above the desired SLA, but the average between Server 1 and Server 2 is bellow the desired SLA.

The second approach views the system under control as a composition of smaller subsystems (servers in the example given above) which are to be controlled. Through the control of each of these subsystems a desired global SLA level can be reached. In the above example, each of the servers would have its own control loop responsible only for that server and which is ensuring that the server maintains a response time bellow the required SLA. Such an approach allows for heterogeneous resources to be part of the same cloud. It also allows for mixed clouds which can contain virtualized and non-virtualized servers running together. Furthermore, requests do not need to be balanced across servers - as long as each server correctly maintains its own SLA even if the cloud encounters a case similar to the previous example where all request of type A come to one server and all requests of type B reach another server the system will maintain a good SLA across all servers in the cloud. An extra advantage of this approach is that it scales very well with even very high number of subcomponents (servers). For the centralized approach if the number of servers increases too much than the central controller starts exhibiting

load and latency issues as well. With a decentralized approach the cloud can theoretically grow forever without suffering degradation due to the control plane. The problem with decentralized systems is that they are more complicated to design, deploy and manage.

There are other approaches used in literature which are somewhere between a fully centralized and a fully decentralized approach, for example hierarchical controllers where subcontrollers take local decisions under the supervision of a parent controller. Such systems will exhibit some of the drawbacks of fully centralized systems but without all the design and deployment problems of fully centralized systems.

One approach which matches well to decentralized systems is that of Self-Organizing systems, as Self-Organizing systems are systems in which a desired state is reached without the use of any central authority or plan. Self-Organizing Systems can achieve high-level goals via the interaction and communication between the components and without the need for a global controller which manages each and all of the subcomponents of the system. From these interactions and communications an emergent system appears which can exhibit self-optimizing features as required by autonomic computing. This approach is the one taken in this thesis due to the advantages that it offers compared to a global controller.

2.2 Related Work

Autonomic computing has become a very important field in research and industry due to the complexity of the IT infrastructure. Autonomic computing is also one of the fundamental technologies required for cloud computing and management of virtualized resources. All autonomic research is based on the MAPE loop in one form or another, however this did not prevent a diversity of approaches to be taken. Multiple architectures have been proposed, as well as various control techniques from machine learning to control theory. This section will be composed of related work in autonomic system architectures, various approaches for the MAPE loop components, self-organizing systems as well as in real-time collaborative systems.

2.2.1 Autonomic System Architectures

All autonomic system architectures are based on the initial proposal made by IBM of the MAPE loop, also sometimes known as MAPE-K loop to include the knowledge portion of the system. However, the use of the MAPE-K loop as a starting point did not hinder a variety of architectures to be proposed in research. The autonomic system architectures which can be found in literature can be split into three main types: decentralized architectures, hierarchical architectures and centralized architectures.

2.2.1.1 Centralized Architectures

Centralized architectures have a common characteristic in that the entire autonomic system - excluding sensors which can run on the measured entity - is running on a single machine which is responsible for running the entire MAPE-K loop and managing all the required resources. Such an approach can be found in the work done at Montana State University [15] where the autonomic architecture is developed as an Autonomic Element. The autonomic element is seen in the paper as the base building block for an autonomic system. An autonomic element is composed of two parts: the manager and the managed element. In the paper, the manager is described as a multi-threaded daemon, with each of the four MAPE functions as well as the effectors and sensors running in their own threads. Message passing is done via queues and synchronization is obtained via semaphores. Figure 2.2 shows a typical centralized architecture for three servers. All server data is measured by one controller, which performs the MAPE process and then takes control actions on the servers.

The architecture uses sensors to obtain data about the managed element and the environment, and defines two separate sensors for each autonomic element - one for internal data from the managed element and an external one for data from the environment. Similarly there are two effectors: one which passes required changes to the external environment and one which modifies the managed element internally. Furthermore, the architecture defines a “data atom” as the basic message that is passed throughout the system. Each atom is an XML data structure that contains version, priority, type and meta-data as well as the base data for the event. Furthermore, multiple atoms can be chained together to form a single atom if required by some common relation.

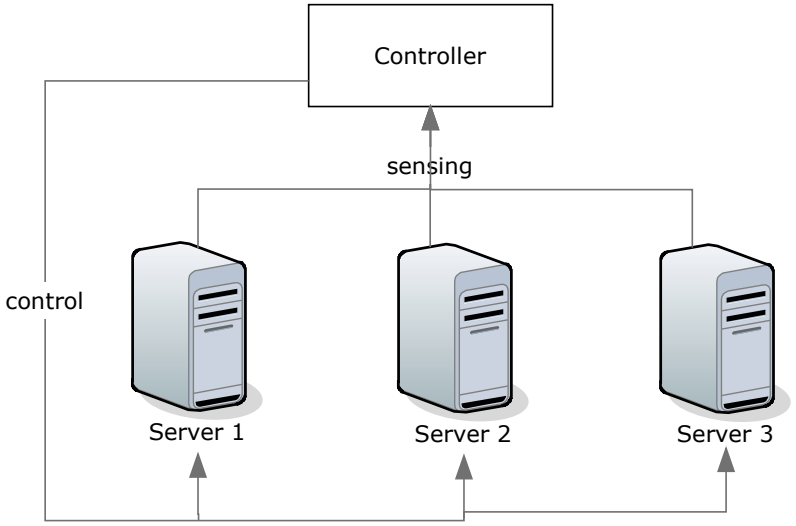

FIGURE 2.2: Centralized Autonomic System Architecture

Since the architecture for an autonomic element uses separate threads for processing, care must be taken to ensure that the system performs correctly asynchronously and that there are no deadlocks in the system. In order to ensure the above, the authors of [15] propose the use of a priority queue for message passing. As data atoms are sent from one component to another, the atoms are stored in a queue, based on their given priorities. The components then pop the respective atoms, process them, and either send them forward to the next component, create new atoms to be sent, or send the atoms to empty data sink to be destroyed. In order to prevent deadlocks and starvation, the authors also use an extra object named a locker, which is similar to a semaphore. A locker is placed between each two adjacent components and its role is to pass atoms between components. A locker receives an atom from a sending component, and notifies the receiving component that it has a new atom available for processing. Since polling is not used, it prevents synchronization and deadlock issues. Lockers are unidirectional, making it impossible to send data back to a previous component for processing, thus resulting in a sequential processing of data. In order to provide rapid response for sensors and effectors, the architecture uses data streams, which are one-way communication channels where one component writes data, and another reads the data in an atomic way. The system also uses a repository to store both processed atoms that are needed later, as well as policy rules. This repository acts as the knowledge base in the MAPE loop.

The above architecture has a number of problems which make it hard to use in

most cases. First of all, the autonomic element architecture executes its entire process on a single machine. This causes problems, which apply to all centralized approaches, in cases where the managed element is a large structure, like a cloud, formed from hundreds of resources. A centralized approach will scale badly as the number of resources which are monitored and managed increases. Furthermore, in the case of geographically distributed resources, the sensor data from the managed resources would have to travel through the internet to the manager, which can add large delays to processing the data and taking corrective actions. Finally, the architecture forces an autonomic element to have two sensors - one external and one internal, as well as two effectors. However there are cases were data comes from a varied number of sources, possible each with its own data sample period, and splitting the data inputs into more than two sensors is desired. Because of the above factors, the autonomic element architecture is restrictive in terms of the kind of autonomic system which can be developed and deployed.

Centralized architectures are also used by some systems developed in industry like IBM's Tivoli Intelligent Orchestrator (TIO) [16] and Sun N1 Service Provisioning System (SPS) [17]. TIO is capable of analyzing the CPU utilization of servers in a cluster and based on utilization over time, and future predictions decide if servers must be added or removed. Once a decision is reached TIO deploys additional servers by finding available servers and based on predefined workflows provisions those servers and adds them to the cluster. A number of issues exist with the TIO platform. First of all, TIO assumes that all the servers in a cluster or resource pool are homogeneous in hardware and have the same behavior. This is rarely the case in data centers and even more so in clouds, where the deployment location could vary. Furthermore even two computers with the same software/hardware will exhibit slightly different behavior. Second of all, TIO can not monitor multiple metrics, and provide multiple provisioning decisions to the same cluster. Only the CPU is monitored and only add/remove server actions are supported in the autonomic mode. Third of all, TIO is designed as a monolithic piece of software making it hard to extend in order to add extra intelligent behavior. Unlike TIO which manages clusters of servers, N1 automates the deployment and configuration of web applications, thus providing the self-configuration characteristic of autonomic computing. Similarly to TIO, SPS is capable of provisioning operating system clones and applications in order to quickly configure a new server.

2.2.1.2 Hierarchical Architectures

Hierarchical architectures improve on centralized architectures by using multiple MAPE-K loops which are organized in a pyramid, with the low level control loops managing fine grained resources, while the high level control loops are responsible for the state of the entire system. Figure 2.3 shows a typical two-layer hierarchical architecture for three servers. Each of the servers has its own control loop and all the low-level control loops have a second layer which manages their parameters.

In [18], the authors propose a three layer architecture formed from bottom to top from: component control, change management and goal management. At the lowest level, the component control layer is concerned with short time scale operations and its goal is to modify the operating parameters of the component under management. At the change management level the system responds to changes in state reported from the lower level. This layer is reactive in that in response to a change in the lower level, it creates a plan to improve or maintain the execution of the system. This layer can, for example, add new components or modify existing components to maintain SLAs. Finally, the goal management layer is responsible for long timescale operations. It takes the state of the system and long ranging goals and produces plans which reach those goals. For example, this layer would be responsible to find available servers which should be deployed in order to maintain a required SLA, or how to migrate servers to maintain a desired redundancy rate after a server's failure. It should be noted that the author's architecture is based on Gat's model for robot architectures [19].

Similarly, in [20], the authors develop a three layer architecture where the system manages from bottom to top: the component, the application and the system. Similarly to the work in [18] the component level manages a single component and takes actions that span a short time. This could be for example a web server which modifies its buffer pool size or thread pool size. By itself, this is not sufficient to manage the SLA of the system and the component level also can not set its own QoS targets. The middle level controller which is the application controller, manages the performance of the entire system by setting QoS parameters for the lower level controllers and taking over when the low level controllers can not reach the required QoS. Such an action could be to redistribute incoming requests to ensure that some servers are not overloaded. The highest layer controller - the provisioning controller - is responsible for rearranging the system when it can

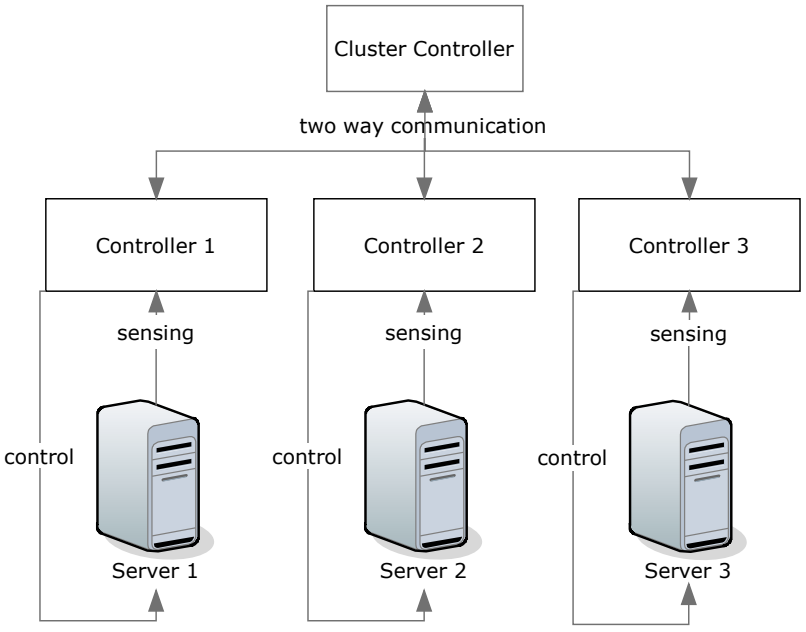

FIGURE 2.3: Hierarchical Autonomic System Architecture

not meet its QoS without structural changes. Such changes can include adding or removing servers to/from the cluster in order to maintain the QoS attributes while also ensuring optimal utilization of resources.

While hierarchical architectures improve on centralized architectures by separating the control concern into multiple levels, where long running self-managing decisions can run at a higher level, while short timeframe decisions can be made close to the component to be changed, some of the issues with centralized architectures still remain. First of all, the use of high level control loops still forces an assumption that the systems components are homogeneous. If the components were to be heterogeneous, then then high level control loops would need to make use of complex models to represent the variation in power of the various components. Second of all, high level control loops would still face scaling problems as the number of components to be managed grows.

2.2.1.3 Decentralized Architectures

Decentralized architectures make use of no global controller and base themselves on the interaction between components to reach a self-managing state. Figure 2.4 shows a typical decentralized architecture for three servers. Each of the servers has its own control loop and the autonomic behaviour is obtained through the

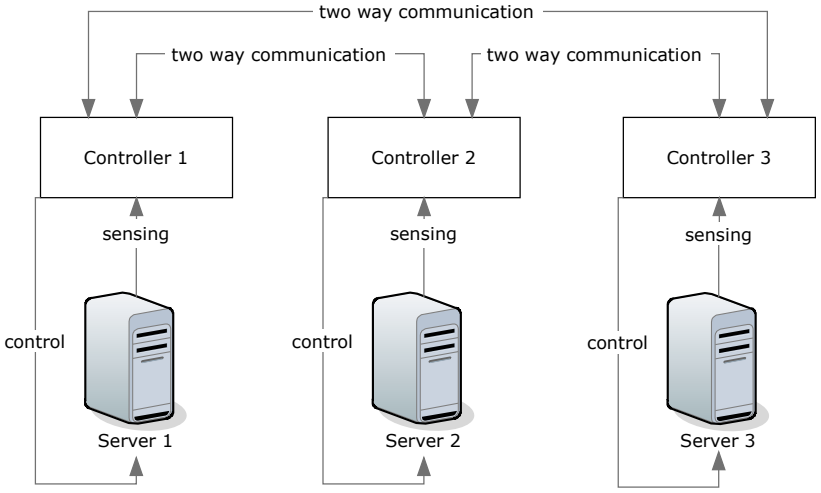

FIGURE 2.4: Decentralized Autonomic System Architecture

communications between the controllers of the servers. Note that not all servers have to communicate with all other servers.

Approaches to create such system range from biologically inspired architectures to self-organizing systems. Such an approach to build autonomic systems is that of a self-managing peer-to-peer architecture, as described in [21]. In this approach each element of the managed resource has its own management logic, and the overall Quality of Service (QoS) is achieved through peer-to-peer negotiation between the elements. While the architecture does distribute the management function across various elements, the architecture is not fully peer-to-peer, the authors choosing a hybrid approach. The top level of the QoS is reached through peer-to-peer negotiations, the lower level state is reached through hierarchical control. The described approach is meant for "the autonomic management of heterogeneous networks and services" [21], but it can also be extended to other autonomic systems.

Another approach for developing decentralized autonomic systems is that of using biologically inspired architectures. In recent years, biologically inspired algorithms like the Ant algorithm [22] for best route discovery in a network have been investigated and developed. In [6], the authors propose an approach which is closer to biological models, than to IBM's vision for autonomic systems. Instead of looking at an autonomic systems as a component that controls another resource, the authors design an autonomic system as a system that is able to change its internal behavior and its external interactions with other components, but is unable to modify external components. In order to reach this design, the authors create a "SelfLet" which is a self sufficient component, able to interact with other SelfLets

in order to reach the high-level goals of the system. Each SelfLet has one or more behaviors, goals and actions. A goal represents a high-level objective, which can be achieved by executing some behaviors. Behaviors are performed by executing a set of local or remote actions. Similarly to the peer-to-peer architecture, SelfLets also have a negotiator manager, which they use in order to negotiate with other SelfLets a way to reach the goals.

Ant algorithms have also been investigated in autonomic systems. In [23] the authors present an ant algorithm to decide where to place services in a cluster. Ants are created and sent in the datacenter to find available servers to use. As the ants traverse the servers in the cluster, they select servers where to deploy the service by examining each servers local state. Similarly, in [24] the authors use ant algorithm to determine loads conditions and move jobs between servers in order to achieve load balancing in a data center.

In [25] the authors use an approach similar to a peer-to-peer architecture. Each autonomic manager which comes online, for example for a new server starting, joins an autonomic management neighbourhood and exchanges messages periodically with members of the neighbourhood in order to determine the state of the system. Based on the information exchanged between autonomic managers, the autonomic managers can then make decisions on how to take actions to optimize the system. The authors apply the architecture to the grid problem of determining where to run various jobs in a grid while maintaining good performance for the various jobs running concurrently. In order to determine where to run the jobs, the system uses a neighbourhood of autonomic managers which decide the machines where jobs should be run, and once started monitor the jobs under their control to know the performance of the system. The managers also exchange information regarding the machines where jobs are being run and the performance of the respective jobs.

The decentralized approach has benefits in terms of scalability with respect to large systems, can be used for heterogeneous clouds and also works very well in systems where components can join and leave the system. As such, for the geographically distributed cloud presented in this thesis, the best approach is a self-organizing systems where each server manages itself and interacts with other servers in order to obtain a global optimal state for the entire cloud.

2.2.2 Autonomic System Approaches

In terms of approaches to creating the intelligent structure of an autonomic system, a variety of approaches also exists. Research into how to develop autonomic systems range from machine learning to control theory and include also policy based decision making, markovian models and economic models.

2.2.2.1 Machine Learning

Due to the fact that autonomic systems attempt to introduce intelligence into the management of the IT infrastructure, machine learning seems to be a very good approach to develop such systems. As such, machine learning methods have been widely used in research in order to create autonomic systems.

In [26] the authors use machine learning approaches in order to predict the performance of database queries using only information available before the queries execute. In order to achieve this goal, the authors examine five machine learning approaches: simple regression, clustering methods, principal component analysis, canonical correlation analysis and kernel canonical correlation analysis. The goal of the prediction was to try and compute, before a query is executed and using only data about how the query is structured and the query plan created by the database optimizer, multiple performance metrics such as CPU usage, memory usage, disk usage in order to be able to better predict resource contention in the database. Furthermore, the prediction had to perform well for both short running and long running queries as well as for a database with a different schema than the trained database. The authors conclusions are that regression does a poor job of predicting the execution of SQL queries, in part because the regression algorithm ignored some of the query features when building the model. In terms of clustering, the authors conclude that for predicting both how long the query will take to execute and the performance characteristics of the query, they would need to run the clustering algorithm separate for each of the two datasets (execution time and performance characteristics). This approach would not find correlations between the two datasets. Principal Component Analsysis (PCA) has the same issue as clustering, as PCA can be applied only to a single multivariate dataset. Canonical Correlation Analysis (CCA) meets the authors goal of being able to find relations between pairs of multivariate datasets, however due to the fact that CCA

uses an Euclidean dot product between the feature vectors of the two datasets to determine similarity the authors conclude that CCA is overly restrictive for the prediction of SQL query performance. Because of these restrictions, the authors use Kernel Canonical Correlation Analysis (KCCA) which is a generalization of CCA that uses instead of Euclidean dot product for similarity measurement, kernel functions. The authors proceed to use KCCA in order to run experiments on a dataset constructed from a range of query types and obtain good results at predicting the performance metrics of the queries.

A number of issues need to be noted about the work in this paper however. First of all, KCCA takes minutes to hours to train and as such the approach can not be used to retrain on the fly while the system is running. Second of all, the way the training dataset is constructed has major implications on the performance of the system as the machine learning algorithm will not be able to deal with anomalous queries which contain situations that did not exist in the training dataset. Finally, as the authors prove, it is very important to analyze the system which is to be managed as different machine learning approaches can be good for one management problem, but not for another.

In [27] the authors develop a system which adds and removes servers from a cluster based on the cluster's workload by using a nonlinear regression model. The authors use a three step approach to predict the new number of servers to have in the cluster. The first step is to use a linear model to predict the next five minutes of workload based on the previous 15 minutes. It should be noted that no motivation is given for the use of 5 and 15 minutes respectively and that is unclear also how appropriate a linear regression is for workload prediction. While an ahead prediction of 5 minutes based on the last 15 minutes could be appropriate for the authors case, the same approach will most likely not work for all deployed clusters. The author's second step is to predict the required number of servers using the predicted workload as an input and a performance model, which is developed as a nonlinear regression model, to predict the number of required servers. Instead of trying to model all the parameters of the system, the authors prefer to detect whenever the model is no longer in sync with the system and create a new performance model from production data. In order to be able to use a new model, the controller of the system acts in two separate modes. In the first mode the controller acts very conservatively as it explores the production system in order to build the model. In this mode, the controller ensures that a very large number of

servers are deployed in order to provide good performance and it slowly removes servers, trying to find the optimum. As the accuracy improves, the controller passes into the second mode called optimal mode in which the built model is used to add/remove servers based on workload predictions. It is easy to see that this approach causes suboptimal behaviour whenever the model has to be changed, as a large amount of extra resources which are not needed will be used. Finally, in step three the system determines how many new servers to add or remove. This is done based on the performance model's desired target but using a hysteresis approach with gains α and β for adding or removing servers in order to avoid oscillations. α and β are selected using apriori simulation runs (the simulations actually set α to 0.9 and find the corresponding β value). The apriori selection of gains can also cause an issue since the model is chosen dynamically and previously selected gains can behave badly with a newer model. The system is simulated during an experiment on a pre-recorded workload of three days. The system manages to maintain the desired SLA (less than 5% of requests have response times larger than 1s) and during the period the controller takes 55 add/remove actions. Looking at the authors performance graphs however, the system appears to be overcompensating while adding servers. Once a maximum number of servers is reached, even though the workload remains constant the system can remove servers safely and not cross the SLA parameters. This is caused most likely by the linear regression used for workload prediction.

Other machine learning approaches are those that use Bayesian models and neural networks. In [28] the authors compare the modelling of response time for a service-oriented architecture via Bayesian networks and neural networks. In order to build a Bayesian model the authors assume that each of the services is independent from all the others, and that the response time for the composition of the services is obtained due to a causal relation between it and the time elapsed on each service. As such, the authors use a Bayesian Network in order to model the systems response time. Bayesian modelling is based on Bayes' theorem which states that "the probability of a hypothesis H conditional on a given body of data E is the ratio of the unconditional probability of the conjunction of the hypothesis with the data to the unconditional probability of the data alone" [29]. In terms of the response time modelling, this means that probability of a certain response time, given a distribution of elapsed times for each of the services is dependent on the prior known response time for the distribution and the conditional probability of that distribution. On the other hand, the neural network model is built as

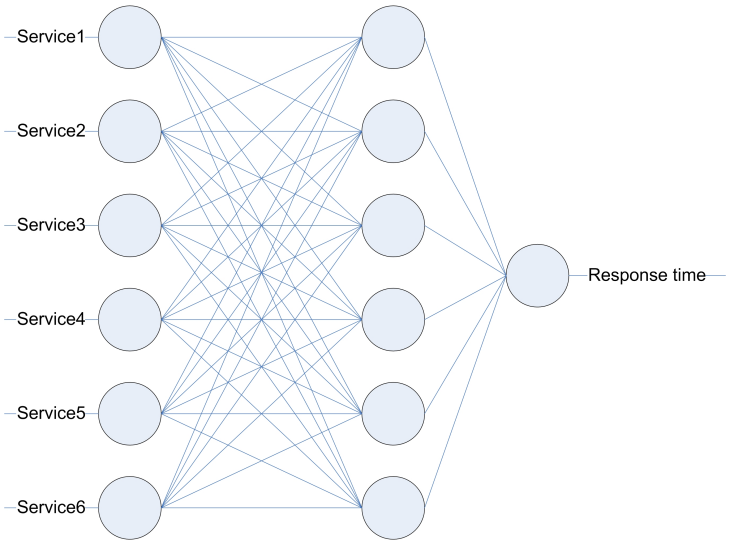

FIGURE 2.5: Neural Network Model

a feed-forward neural network, with six input nodes, six neurons in the hidden layer, and one output layer with one neuron. Figure 2.5 shows the structure of the neural network. In the neural network model, the various weights of how the inputs affect the hidden layer, and how the hidden layer affects the outputs must be found. Even though they are built differently, the two models have a number of common operations. Both models use newly measured data, both input and output in order to learn the system's behavior and improve the model. Thus the model must be able to receive new data and update itself. Furthermore, the models must be capable of being used in order to predict the future state of the output variables based on the the current state of the observed variables.

Similarly in [30] the authors use a Bayesian network in order to predict the response time of a database query or workload using a model which is built online and can be modified at run time. In order to achieve these goals, the authors use a Gaussian based response time model to encode the Bayesian probabilities. The Gaussian model is first trained offline and is capable of adapting itself once running online. Using a number of Gaussian models (linear and non-linear) the authors perform experiments on a database test bed and show the performance and robustness of the various models. One thing to note from the results is that all models have problems predicting queries that have not been seen before. Similarly, the models have problems making predictions when the online system is configured differently from the training system.

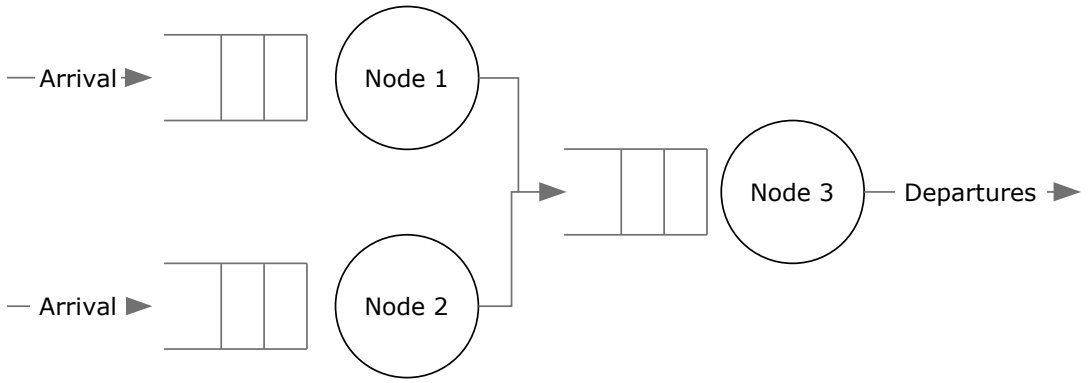

FIGURE 2.6: Layered Queueing Network Model

2.2.2.2 Layered Queueing Models

A different approach taken to develop autonomic systems is to use Layered Queueing Models (LQM) to represent the behaviour of the system, and then to solve the LQM to predict the future state of the system. Autonomic Queueing Models base themselves on Queuing Network Model (QNM) theory, which sees an IT system as a network of queues such as CPU queues, I/O queues, network queues, etc. Requests to the system are split into a number of classes, where each class of requests exhibits the same behaviour with respect to the queues. The response time and throughput of the system can then be calculated based on the time it takes for the requests to go through the queues. A very important note about QNMs is that the structure of the queues can not be changed at run time and represents a static view of how the system must behave. As such any online adaptation in a QNM can only change the various parameters for the queues, but not the underlying structure of the queues and connections between queues. Figure 2.6 shows a Layered Queueing Network with 3 nodes (each node has its own queue) in two layers. Arrivals into the system come into nodes 1 and 2, and from there requests move to node 3 from which the output is dispatched. Such a model can be used for two application servers with one database for example.

Such an approach is taken in [4] and [31] where the authors develop queueing models for multi-tiered transactional applications (i.e. a web server facing the client and sending requests for processing to an application server which uses a database to store and retrieve persistent data). In [4] The LQM developed has a number of roles. First of all, the model is used to predict the future response time for each of the request classes by first predicting the future number of clients arriving in the system. If any of the predicted responses is outside the desired

SLA, use the model to compute the number of servers which would alleviate the bottleneck since any high response is caused by bottlenecks at one or more of the queues. The new number of servers is computed using a hill-climbing algorithm. Finally, provision the new servers that alleviates the bottleneck problem. One thing to note, is that "the forecasting step is in the order of minutes", however there is no mention of what a good time period for forecasting would be or how various forecasting periods impact the forecasting accuracy. The problem in [31] is similar in that an LQM is build for a multi-tiered system. However, the authors develop a heuristic to try and maximize the profit obtained by the difference between revenues from SLAs and the cost of keeping servers on. This is done in part by load balancing when possible multiple similar requests to the same machines and by deciding which servers to allocate to which application tier. The authors approach is used to show results that suggest better performance than when using a proportional assignment schema. One issues that exists with this approach is that the algorithm takes a long time to find the maximum, and as such can only be run periodically and can not be used to continuously improve the system. At the same time, the authors choose a value of 60% utilization as good utilization for the servers, but autonomic systems should attempt to obtain higher utilization in favor of using less machines, since it is preferable from an economical point of view to use one machine at 100% than to use two machines running at 50% each.

While the high-level modeling approach for the QNM is very similar to the Bayesian models described previously, the differences appear in how the model is built and used. The Bayesian models output is based on the previously known probability of an observed distribution (past information on behaviour leads to the prediction of future behaviour), while the QNM's output is based only on the current size of the queues (present state of queues predicts future behaviour). The similarity between the models appears again regarding the functions that they perform. Both models are used in order to predict the future state of the system's output variables based on the current model state and a new input value, and both models can update their parametric model based on observed input and output values.

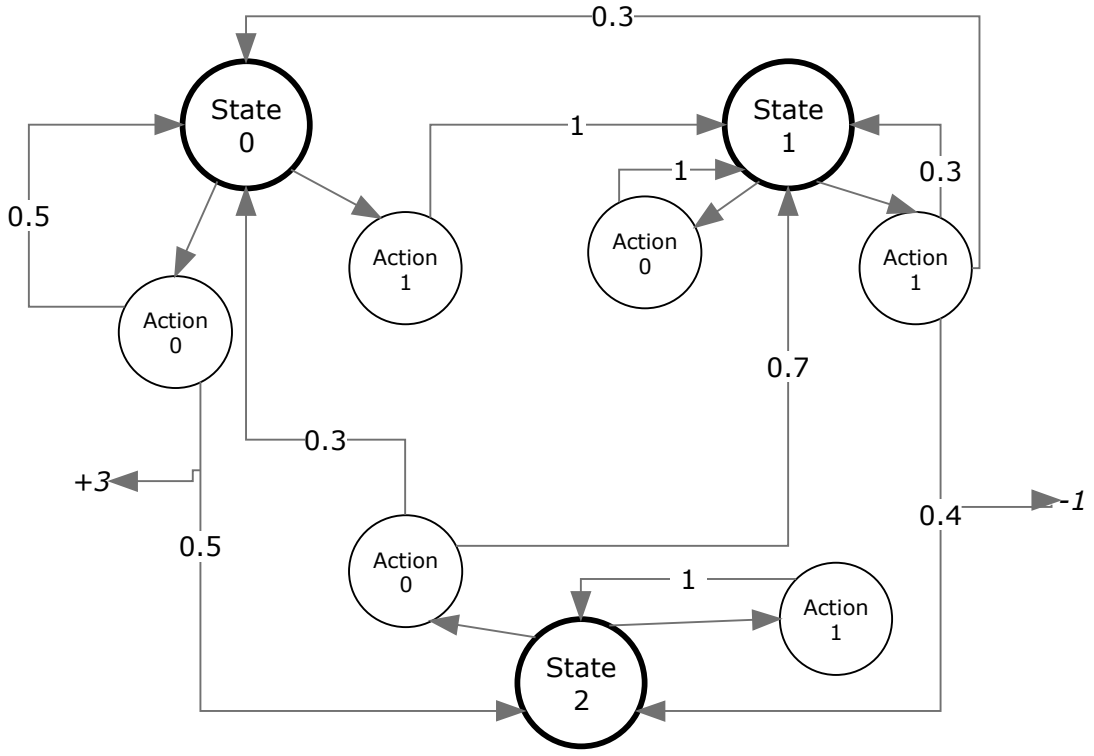

FIGURE 2.7: Markov Decision Process Model

2.2.2.3 Markov Chains

A third approach, which appeared more recently in literature is that of using Markov chains, either discrete or continuous, in order to model the behaviour of the managed IT system. Figure 2.7 shows an example of a Markov Decision Process with 3 states and 2 actions. Each of the states can lead to any of the two actions, and from each action a certain probability leads to a different state. Some of the transitions have either positive or negative rewards, seen as the italic values next to some transitions.

In [32] the authors introduce a framework for autonomic systems which uses Markov chains in order to model the system and by quantitative analysis of the Markov chain make decisions on how to adjust at run time the parameters of the IT system. In order to develop the system, the authors employ a probabilistic model checker/quantitative analyzer which uses a cost/rewards model to improve the parameters of the system. The autonomic system is applied to two autonomic problems.

The first problem is that of dynamically managing the power of a hard drive by switching the state of the drive to sleep when it is idle. In experiments with

the autonomic system, the management obtained using Markov chains performs better in terms of utility when compared to a simple timestamp method and N method. Also to note, the execution time of the autonomic system is sub-second but the CPU overhead is in the 1.5% to 2.5% range which is high for a problem like disk management. The second application of the autonomic system is for the adaptive management of a group of servers in which servers can be assigned to one of multiple clusters with clusters having different levels of SLA. The desired goal is to assign servers in such a way as to always reach the target availability of the most important cluster and trickle down in terms of cluster importance. For this purpose, a Markov chain is created for the cluster as well as a utility function which is to be maximized. A simulation is run with the described system and while the autonomic management ensures that each of the clusters has sufficient servers, the system does not deprovision unused servers. As such, the clusters have more servers than required. At the same time, based on the given results the system over-provisions the highest priority cluster by a large margin. In terms of execution, the authors mention that the system takes 30 seconds to compute the new state, which is an acceptable time to take, since provisioning servers takes an order of magnitude larger time to execute. A further issue is that it is unclear from the paper, if the system is reactive (changes the system once the SLA is breached) or proactive (predicts an SLA breach and changes the system before the SLA breach happens).

Similarly, in [33] the authors use a Markovian decision process policy with reinforcement learning in order to perform resource allocation in a cloud environment. Markovian decision processes are an extension of Markov chains which allow for the existence of multiple actions for a state, thus including choice while going from one state to another, and of rewards, thus including a way to motivate the selection of a certain path. The goal of the system, is to manage a cloud datacenter, such that VMs are allocated in such a way as to maintain a required SLA for each of the applications running in the cloud while at the same time optimizing the number of VMs for each application. This problem is represented by the authors in the form of a Markovian decision process which takes into consideration the workload, the number of VMs allocated and the response time. Based on these values and the state of the decision process the system decides how many VMs to add or remove. One issue with a simple decision process is that the probability distribution functions are computed offline, and the system simply uses the predetermined functions to compute an optimal policy. The authors alleviate this

problem by using a reinforcement learning approach to update the probability function by using rewards for good decisions. The authors apply the system on a simulation of a cloud with one application. The Markovian decision process system is capable to reach a policy which ensures that the SLA is maintained quite quickly. However, it is unclear if this allocation is optimal, since in the graphs the response time is 20 - 40% lower than the required SLA, which could simply be due to over-provisioning of VMs. One interesting note of the authors is that long running systems will exhibit major changes in behaviour when the model has to be changed entirely since a reinforced learning system will not be able to react promptly due to the "old" knowledge in the system. At the same time, the authors note that the information used as input for the system is insufficient for appropriate decision making in the real world, as more data from the environment will be needed.

2.2.2.4 Economic Models

A similar approach to a punishment/reward system in order to make decisions is that of using economic models in policy making. Economic models consist of two actors: suppliers and consumers which use a trading mechanism to exchange resources. The trading mechanism is usually based on some form of "currency" which consumers use in order to obtain resources from suppliers. In [34] the authors apply an economic model in order to translate between business policy and low-level requirements for the automation of a database system. The control goal of the system is to tune the buffer pool of the database as well as the CPU usage when faced with multiple workloads. The workloads act as consumers and each workload has its own buffer pool and its own set of database agents, but the system does not have enough resources for all the workloads to execute quickly. Each workload has an estimated wealth based on the relative importance of the workload and the estimated work required for the workload, which is calculated based on the database's query optimizer prediction of I/O operations for the query. Using this wealth consumers (workloads) bid on resources dependent on the marginal utility provided by the given resources until either no more resources are available, the consumer uses all its wealth or the consumer has sufficient resources. A resource broker is used to enable the bidding process. The resource broker divides all the resources into blocks of resources which are up for bidding. For each of the resource blocks, the resource broker receives sealed bids from the consumers and

chooses the highest bid. The bidding process repeats until no more resources are available, no more resources are desired by consumers or consumers do not have any wealth remaining. The above system is validated by running tests on an experimental environment. The economic model is first used to determine the resource allocation offline and the database is configured based on the results of the economic model. It should be noted that the system is not used to update the database's configuration online based on changes in workloads. The authors observe that the developed system is capable of obtaining effective allocations which result in similar number of reads to a system configured manually, which respect the desired importance of workloads and which result in the expected relative performance with respect to the relative importance values. One problem which the authors observe with the system is the fact that a real system would not know ahead of time the possible workloads (if such knowledge was available ahead of time manual configuration would be sufficient) and as such the utility functions would be hard to determine at run time. To create a really autonomic system a feedback loop would be required to obtain statistics from the running workloads and update the utility functions.

Another solution for autonomic systems is based on cost functions and optimization of such functions. In [31] which also uses a Layered Queueing Model and was described previously such a method is described. The authors assume that in a data center there are various costs: energy, hardware, and software utilization, as well as revenues obtained by providing the service and penalties due to breaches of SLA. The plan function is then based on the observation that "revenues increase almost linearly with the system load and start decreasing after a maximum that is obtained when the service center utilization is about 0.5 to 0.6". Because of this, the plan function attempts to optimize the cost function but since the optimization problem is NP-hard, the solution proposed is a heuristic search which results in a local optimum, but not necessary the global optimum. One important aspect of this paper is that it shows how multiple methods can be combined to create autonomic systems. In this case an economic based policy on top of a queueing model of the datacenter.

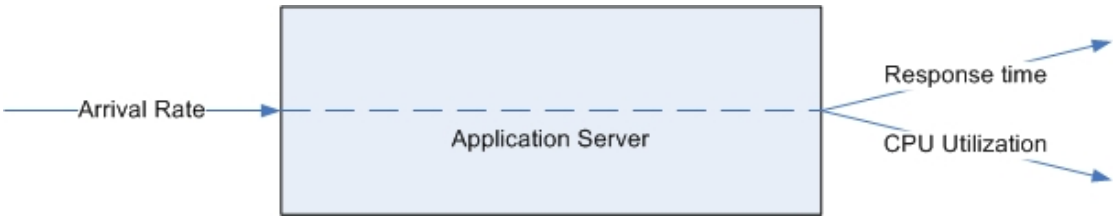

FIGURE 2.8: Server Black Box Model

2.2.2.5 Control Theoretic Models

Due to the fact that autonomic system architectures are based on the MAPE-loop an approach to develop such systems which maps extremely well on the MAPE loop is that of using control theory methods and algorithms. One very important aspect of developing a system based on control theory (which also applies to some of the other methods) is that of creating a model for the system. Figure 2.8 shows an example of a black box model for an application server. The arrival of clients acts as a perturbation on the system, which will change the internal state of the system which is hidden by the black box and in response to the state change the outputs of the system - CPU utilization and response time will be modified.

The authors of [35] explore how control theory can be applied to systems research in order to develop a management system for cloud computing environments. The authors note that such an environment, in which applications are consolidated across virtual machines and share dynamic resources based on the changing workload for the applications, pose a number of challenges to system administrators in the form of: "complex SLAs, time-varying workload demands, distributed resource allocation and resource dependencies". These challenges are addressed by using control theory in order to model, analyze and design a feedback loop for a resource management system. The autonomic system developed is applied to the problem of resource allocation in a shared virtualized infrastructure. The system does not provision new servers for the various applications, it just changes the resource allocations for the various virtual machines dynamically. The goals of the system are to guarantee application performance by reaching all the desired SLAs for the applications if possible (high demand periods could cause all resources to be used and low priority applications to miss their SLAs) and to provide high resource utilization and performance differentiation during contention periods (high priority applications maintain their SLAs over low priority applications). The authors

identify six strengths of the control theoretic approach for autonomic systems, which are:

1. Control theoretic systems offer a quantitative input-output model where a black-box model can be developed by performing offline experiments in order to determine the relation between the given control input, the perturbations in the system and the observed outputs which are the values to be controlled. Use of a proper input-output model in relation with a feedback controller offer the advantages of ensuring that the system converges towards equilibrium, that the system is stable and converges in appropriate time and that the controller can adapt to different regions of the input-output relation (for example if the input-output relation is bimodal).

2. Control theory offers the ability to study system dynamics, which refer to the transient effects of past states on the current system state. Such input-output models can also be built by using system identification experiments to determine the degree of correlation between current output and past output values.

3. Furthermore, using control theory system designers can develop multiple-input and multiple-output (MIMO) control models in order to capture the relations between the various inputs and outputs of the system. The relations in such systems are hard to quantify even when they are known.

4. Control theory provides a number of well researched and well investigated algorithms for control based on models discovered via system identification for both single-input, single-output (SISO) and MIMO systems.

5. Control theory also offers methods to determine the stability of the closed-loop model and control system. At the same time, control theory deals with the tradeoff between stability and responsiveness of the control system which is the tradeoff between how fast the system reaches a change decision and how stable the control system's decision is. A system which reaches decisions too fast will oscillate and be unstable as any small change in perturbations will result in a control decision. On the other hand a very stable system will make changes very slowly and miss important changes in perturbations.

6. By using adaptive control theory, system administrators can develop controllers for systems with varying behaviour over time. This can be done by

estimating the parameters of the model online and attempting to keep the parameters in line with the actual state of the system.

For each of the six strengths the authors offer examples of how they applied control theory to various autonomic problems. The authors however, also note some limitations of control theory which must be taken into account while developing autonomic systems. First of all, computer systems exhibit non-linear behaviour which makes modelling more difficult. Second of all, the development of black-box control models requires very controlled experiments to be run. Such experiments can not be usually run based on data from production systems since such data rarely contains the required perturbations or constraints to determine the relations between input and output. Third of all, classical control theory deals with continuous inputs. While discrete values can be quantized, this can lead to instability or inefficiency. Finally, classical control theory usually deals with tracking problems - the control system maintains a variable at a certain value. For autonomic systems, the goal is generally to maximize or minimize some values (maximize resource usage while minimizing response time for example). Any person who creates autonomic systems by using control theory must take these limitations in consideration and deal with them.

Similarly, in [36] the authors compare various approaches to creating decision makers for autonomic systems. They look at heuristic approaches, standard and advanced control-based approaches as well as model-based and model-free machine learning approaches. In order to compare the five methods the researchers look at a standard autonomic computing problem, namely the problem of resource allocation to some running application in an operating system. The application which is assigned resources has a way to signal "heartbeats" at important places in code as well as to describe desired performance goals. The authors conclusion is that an adaptive control approach like a Kalman Filter produces good results for a range of applications, and this matches the observations of [35]. The authors note that for a given system with a narrow range of behaviours testing multiple approaches is the best approach, for a system with a broad range of behaviours or even unknown behaviour ahead of time (a self-optimizing application server with unknown applications running on top) adaptive control is the best approach.

The issue of applying control theory methods for managing the CPU shares received by multiple resource containers running on the same machine is addressed

in [37]. Resource containers are a way to provide performance isolation for applications running on the same hardware and they can be either an operating system feature or an entire technology like virtualization. Usually the resource container resource allocation is done offline before deployment and does not change during the lifetime of the applications. The authors present an approach to change the resource allocations dynamically by using a feedback control loop based on control theory principles. The specific application under control in this case is an Apache Web server running inside HP-UX PRM as a resource allocation system. The model of the system is created by applying system identification on observable metrics for a black-box model. The reason to use a black-box model given by the authors is that the complexity of a web server makes it hard to obtain a control model from first principles. The model for the system is a SISO model which correlates the CPU usage of the resource container to the Mean Response Time (MRT) of the web server. The system identification is done using a single workload meant to stress the CPU of the container. Using the model obtained by system identification, the authors apply a simple Proportional-Integral (PI) controller and show that such a system is stable, has a rise time of 3 sample periods and settling time of 6 sample periods. The rise time refers to the time it takes for the controller to respond to a change in perturbations, while the settling time refers to how fast the controller reaches stability after a perturbation.

The problem with the PI controller is that it is "trained" through system identification on one workload and would not respond very well to a different workload. In order to deal with unknown workloads the controller must be developed using adaptive control. This observation matches that of [35] and [36] which note that adaptive control is best for systems with unknown behaviour ahead of time or for those with changing behaviour. The adaptive control is applied to the system by adding a parameter estimator for the model discovered using system identification. The two parameters are estimated using the recursive least-square (RLS) method and are updated at each interval. The parameters are then fed into a pole placement module which chooses the desired parameters of the PI controller in order to have the poles of the transfer function at the desired location. The two control systems (adaptive and simple PI) are evaluated on a test-bed. The simple controller is capable of maintaining the response time within 15% of the target value when being faced with a workload similar to the one used for system identification. The adaptive controller is capable of maintaining the response time within 20% of the target value even with changes in the workload. However, the

tradeoff is that the convergence towards the desired response time is slower using the adaptive controller and that this controller also oscillates more. The simple controller is not tested on the varied workload since its results would be bad.

[38] deals with a similar issue, that of developing a feedback control system for managing the service delay (response time) for web servers based on relative or absolute delay guarantees. Absolute delay guarantees require that each class of services is served within the desired time when the server is not overloaded. When the server is overloaded admission control is applied based on some form of priority order over the services. In relative delay a fixed ratio between the delays of the various classes is maintained. The problem with ensuring such guarantees is that the workload is unknown ahead of time, and as such developing a robust control system for service delay guarantees is complicated. Two types of controllers are used in the proposed system - one for relative and one for absolute service delay. The absolute delay controller uses a PI controller similar to the work in [37] and the control parameters are determined using system identification. The relative delay controller computes the relative delay between two adjacent classes. Thus the system contains n-1 controllers for n classes of services. Each of the controllers uses a PI controller with extra logic on top to calculate the ratio of the two service classes under control. Similar to all the other control theoretic papers, the authors note that such a system would face problems if the workload during deployment is vastly different from the workload on which system identification was done. In order to alleviate this issue one can use adaptive control. The system is applied to a testbed of Apache Web Servers for both absolute and relative delays with two and three classes. The experimental results show that the control method is capable to reach the desired guarantees when run in a closed-loop, while the open-loop solution violates the desired delays when the workload changes.

Similar work is performed in [39] where the issue of managing the database connection pool in a web server is approached from a control theoretic point of view. The goal of the work is similar to that of [38] in that the system's final goal is to improve the response time of a server under heavy load. Unlike [38] this is achieved by setting priority for the idle connections in the database connection pool, however similar to [38] this is done by using the dual approach of proportional delay differentiation and absolute delay differentiation. Like [38] the authors use system identification to determine the PI controller's parameters by applying

the RLS method. The system is tested on a server running Tomcat and the system is capable to maintain the desired response time in the face of varying workloads.

Based on this work a number of observations about control theory for autonomic systems can be observed. First of all, all research in autonomic systems control theory has focused on developing black box models due to the difficulty of applying first principles to IT systems. Second of all, most researchers focus on using simple controllers like the PI controller for autonomic systems and use experiments and system identification in order to find the parameters of the controller. In order to deal with varying perturbations, researchers apply adaptive control on top of the simple PI controller.

2.2.3 Self-Organizing Systems

Self-Organizing Systems deal with complexity in management systems by allowing the management of the system to appear from the interaction between the components of the system. Such systems have no global controller and components do not have access to information outside their own local information. Components can however exchange messages with each other to inform other components about the sending component's state. Through this, "emergent" behaviour appears in systems, behaviour which is not imposed from outside the system. Like autonomic systems, self-organizing systems are inspired by behaviour observed in nature like "synchronous flashing that sometimes develops among aggregations of thousands of fireflies in southeast Asia" or the stripped and mottled pattern which appears in zebras and fishes. Experiments and theoretical work suggests that such apparent complex patterns appear from a small number of very simple rules and research into them can be found in [40], [41], [42], [43], [44], [45], [46].

The problem of the stripped pattern was examined in computing by Alan Turing who first suggested in 1952 the general schema for this mechanism of self-organized pattern formation. The self-organizing system has a number of sites which act as activators. The activators produce two pheromones: a short range activator which enhances the activator production and a short range antagonist which ensures that the self-enhancement is localized. By performing simulations using a model of such a system both stripped and mottled patterns can be obtained. This shows that complex behaviour can emerge from very simple rules through the collaboration of components.

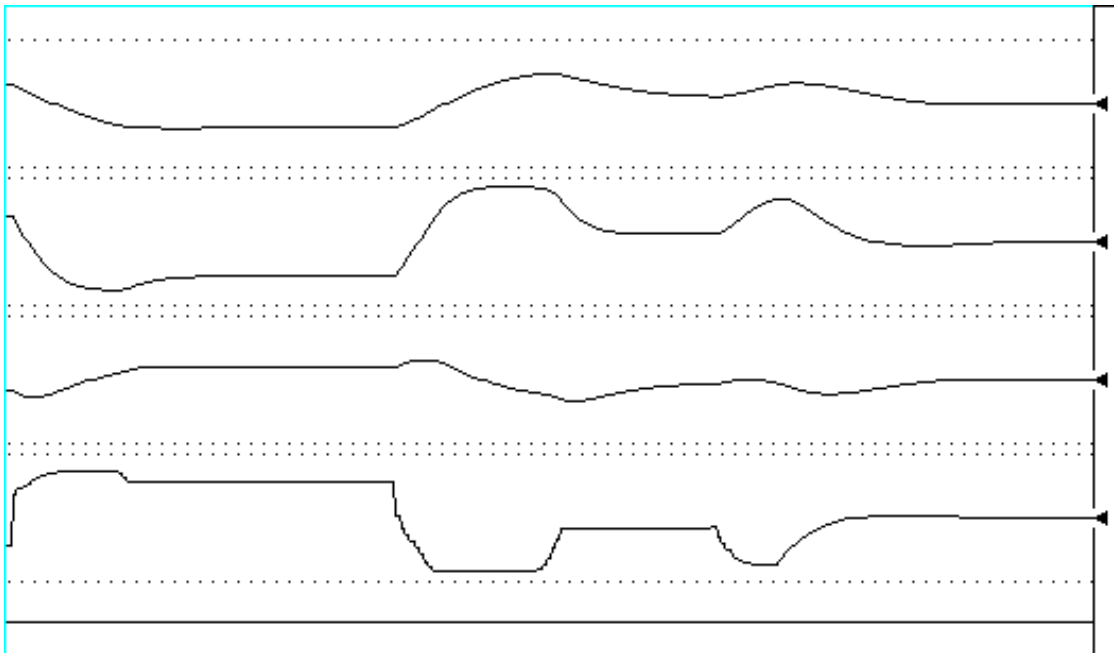

FIGURE 2.9: Homeostat

Self-organizing systems were also employed by Ashby to create his homeostat [8] which is capable of adapting itself to any perturbation in the system and reach back a stable state. The initial system built by Ashby was an adaptive ultra-stable system with the goal of showing Ashby's law of requisite variety, which states that a control system needs as many internal states as those of the system being controlled. The system automatically adapted its internal configuration to stabilize any disturbance introduced in the system. Figure 2.9 shows the output of a homeostat. As any of the four pens is perturbed, the system self-organizes until all four pens are back to a stable state.

In [**?**], the authors look at a number of self-organizing algorithms and compare them in the context of the problem of load balancing requests in a cloud environment. Some of the algorithms examined include Particle Swarm Optimization, Artificial Bee Colony, Genetic Algorithms and Ant Colony Optimization. Furthermore, the authors examine in detail the Ant Lion Optimizer algorithm for load balancing due to it's better performance and better ability to search the entire solution space instead of falling into a local optima. The authors observe that the self-organizing algorithms perform better than traditional algorithms in terms of quality of service provided and minimize the response time, at a tradeoff of more network and processing overhead.

2.2.3.1 Genetic Algorithms (GA)

One of the oldest approaches for self-organizing systems is based on mutating solutions and maintaining only the fittest solutions to continue mutating in the future. The optimization usually starts with a random solution which is encoded in some form of digital representation - usually a binary string. There also exists some form of fitness function which decides how good a solution is. Solutions are chosen in pairs and they exchange part of the solution to form two children. Mutations can also be introduced by randomly changing part of a child. After the population of children is obtained, the children are measured using the fitness function and unfit children are removed from the population. The surviving children become the new parents and the loop is repeated either for a given number of iterations or until a certain fitness value is passed. The final solution is the one with the best fitness of the entire population. The pseudocode in 2.1 shows the behaviour of GA algorithms.

Algorithm 2.1 Genetic Algorithm Pseudocode

```
Random generation of initial parents
while end criteria not met do
    Select parent pairs and cross over chromosomes
    Randomly mutate children
    Compute children fitness
    if child is fit then
        keep child in population
    else
        remove child
    end if
    replace parents with children population
end while
return best fit child
```

In the case of the problem presented in this thesis - that of optimizing the size of a cloud of servers dynamically at runtime - a GA could encode the servers to be used as a binary array of 0 or 1, and then find the best solution in terms of future performance of the system by predicting which servers should be up (a 1 in the binary array) or stopped (a 0 in the binary array). The GA can not be used in order to determine when the system is about to breach its SLAs, and can only be used in order to obtain a new optimal state. At the same time, GAs are known to be slow in order to obtain a good solution as the number of iterations taken to

converge can be very large if a good solution is desired. As such, GAs are not a very good fit for a real time self optimization of cloud resources.

2.2.3.2 Particle Swarm Optimization (PSO)

Particle swarm optimization uses as inspiration the behaviour of flocks of birds or swarms of fish looking for food. Each possible solution represents a particle in the search space with a position, and the particles move based on their own speed and the best known position. At the same time, there exists an optimization function f which takes as value the position of a particle and returns a value representing how good that solution is. As such, each particle is initialized with a random position and speed. Each of the position computes it's f function and save it as the position's best solution. The best of the f values is stored as the swarms best solution. At each step of the iteration, each of the particles computes a new speed and a new position based on the previous position and speed, such that each particle's movement is guided by it's own best solution and the swarms best solution. Once an exit criteria is reached - either number of iterations or the desired function is sufficiently improved, the algorithm exits and the best swarm position is used as the solution of the optimization problem. The pseudocode in 2.2 shows the behaviour of PSO algorithms. $f(swarm)$ is defined as the best solution for the swarm.

For example, a PSO is used in [**?**] in order to obtain the initial placement of tasks in a cloud. The system described in the paper does not perform any real-time optimization and only takes care of the initial problem of placing N SaaS tasks on M cloud instances, where $M < N$. The system uses a position vector to describe which task is assigned to which server and then uses the PSO algorithm to find the best placement.

Similarly to the GA algorithm, the PSO algorithm can not be used in order to detect when a system breaches it's SLA and can only be used once some other system detects a possible SLA breach. PSOs can also fall easily into local optima depending on how well the parameters of the problem are set at the beginning. If the parameters cause too small speed/position changes than the system can fall into local optima easily. If the parameters cause large changes then the optimization problem can fluctuate.

Algorithm 2.2 Particle Swarm Optimization

```
Random uniform generation of initial positions and velocities in search space
for all Particles do
    Compute particle's position and save as best position
    if f(particle) > f(swarm) then
        update best(swarm) = particle
    end if
end for
while end criteria not met do
    for all Particles do
        Compute particle's new speed
        Compute particle's new position
        if f(particle_new) > f(particle_best) then
            update particle_best = particle_new
        end if
        if f(particle_new) > f(swarm) then
            update best(swarm) = particle_new
        end if
    end for
end while
return best(swarm)
```

2.2.3.3 Artificial Bee Colony (ABC)

The artificial bee colony algorithm [**?**] is inspired by the process used by a honey bee swarm in order to forage for food. Unlike in other self-organizing systems, not all the composing parts are the same in the ABC algorithm. Three types of bees are used in the algorithm: employed bees, onlooker bees and scout bees. The problem to be optimized is defined in terms of food sources, with different solutions representing different food sources. For each source of food there is one employed bee which goes to that source, forages and returns home. Onlooker bees stay at the nest and establish new food sources based on the information received from employed bees returning to the hive. Onlooker bees choose new food sources by watching the employed bees come back to the hive in the dancing area, and perform a dance based on how good the food source is. Finally, scout bees establish new food sources via random searching around the hive.

Algorithmically, this behaviour is represented by an iterative loop over three phases: employed bee phase, onlooker bee phase, scout bee phase. Initially, the food sources are initialized by having the scout bees go and find random solutions in the solution space. In the employed bee phase, employed bees generate new

solutions by going to an existing solution and searching the space around it for a new solution. Once a new solution is generated, the fitness of the new solution is computed and a greedy algorithm used to choose between the new solution and the original source. Employed bees then go to the hive and share the information with onlooker bees in the dancing area. In the onlooker bee phase, onlooker bees choose their food source based on the information received from the employed bees. Once an onlooker bee chooses a source, it also determines a neighbour and computes the fitness of the new solution and chooses between the new solution and the old. In the scout phase, employed bees which can not improve on their solution after a number of iterations give up and become scout bees. Scouts search for solutions randomly.

The pseudocode in 2.3 shows the behaviour of ABC algorithms.

Algorithm 2.3 Artificial Bee Colony

```
Initialize solutions randomly using scout bees
while end criteria not met do
    Employed bee phase
    Onlooker bee phase
    Scout bee phase
    Store best solution up to now
end while
return best(solution)
```

Like the GA and PSO algorithms, this algorithm can not be used to detect problems in the system, and can only be used to obtain an optimal solution to the problem at hand. Compared to both GA and PSO, ABC offers similar or better performance but with less control variables needing to be used - thus creating a simpler algorithm to implement.

2.2.3.4 Ant Based Algorithms

Similar to the ABC algorithm, the ant colony optimization (ACO) algorithm is inspired by the behaviour of insect hives - in this case ants. The behaviour which inspired this algorithm is the way in which ants can build an efficient path between the nest and a food source using random search and reinforced learning. In real life, ants randomly search for food around the nest. When an ant finds a food source it returns to the nest with some food while laying a pheromone trail. Other ants which meet the pheromone trail are more likely to follow the trail then continue

searching randomly. If the food source is large enough eventually a large number of ants follow the same path as each ant lays down a pheromone trail causing more ants to follow it. At the same time, the pheromones dissipate over time such that once a food source is depleted and ants no longer return with food from it, the trail will disappear.

The ACO algorithm is very similar to the behaviour of ants in real life and is best used to find optimal paths through graphs. For example, the algorithm has been used to provide reinforced learning for routers in a network for better packet forwarding **??**. In simple terms, the ACO is composed of a number of ants which are traversing the network by moving from node to node across edges. As ants move through the network they deposit pheromones on the edges based on some function. When ants decide the next node to go to, they choose based on the pheromones on the nodes going out from the current node. The selection of the next node is done pseudo-randomly such that not all ants follow the same path, however more ants will prefer edges with higher pheromone levels.

In the case of routing in a network, the nodes are the routers and the edges are the connections between the routers. Ants are packets being sent between the routers and either they wait in a normal queue or have higher priority than normal packets. If the ants simply wait in the queues to be processed then they can measure the processing latency at the routers. As ants traverse the network they can update the routing table with information regarding how loaded is the network and also the best routes between different nodes. The pheromones in this case act as a way for more ants to go through low loaded network links such that those links are used by normal traffic.

The pseudocode in 2.4 shows the behaviour of ACO algorithm from the perspective of an ant.

Algorithm 2.4 Ant Colony Optimization

```
while end criteria not met do
    Find next node based on pheromones/random
    Go to next node
    Update pheromones for the edge just traversed
end while
return best(solution)basedonpheromones
```

Another self-organizing algorithm is based on the behaviour of ants when relocating to a new nest in the case when the old nest was destroyed. This behaviour is

similar to that of honey bees when relocating [**?**]. When their nest is destroyed, ants start looking for a new nest by doing tandem runs. In such a case, there are two types of ants - scouts and non scouts. Scouts start to search for new nests randomly and upon finding a new possible nest they determine how viable the new nest is and return home. Upon coming back home ants recruit from the non-scout ants and do tandem runs together. At the same time scout ants can be recruited by other scouts who have a better nest as their new nest. This process is repeated until a quorum of ants has chosen a new site for the nest. Ants can give up on their possible new nest if the population of the new nest starts decreasing - suggesting that ants from this new nest have been successfully recruited by a better nest. The pseudocode in 2.5 shows the behaviour of the ant relocation algorithm.

Algorithm 2.5 Ant Colony Relocation

Round 1: Ants find new nests
while more than one nest **do**
    Round 2: Viable ants return home and recruit from the other ants
    Round 2: Not viable ants wait one turn
    Round 3: Tandem runs to new nest. Upon arrival at new nest ants compare previous and new nest size. Nest with decreasing pop become not viable.
    Round 3: Not viable ants return home
end while
return *singlenest*

The main reason to use self-organizing systems for autonomic system development is that the two have a number of characteristics in common like the ability to deal with unexpected changes in the environment. Unlike the other self-organizing systems presented, the ACO algorithm can be used to detect when the system breaches or is about to breach it's SLAs. As ants move around the system, they can take measurements and approximate the state of the entire cloud based on their knowledge of the pheromone levels across the cloud. Once a certain threshold is reached ants can trigger a request for self-optimization which is computed by a separate algorithm. Because of these reasons, this thesis will use the ACO algorithm to develop the self-organizing, self-optimizing system.

2.2.4 Real-Time Collaborative Systems

An important issue while dealing with autonomic systems, is the system which will be under control. Different systems have different goals and objectives which

can be optimized. While almost all self-optimization systems deal with some form of improving the response time as seen by the user, the way in which such an optimization is reached depends on the actual system to be controlled and the parameters which can be modified. For example, a database server could deal with improving query response time by simply adding more copies. Such an approach however, adds the extra problem of making sure all the database copies have the same data which is not a trivial problem to be solved. A more intelligent solution could be to modify the database's buffer sizes at run time depending on the queries which are being executed. Such an approach can not be taken for an application server however. Because of this, it is very important to understand both the requirements for self-optimization in a system, as well as how those self-optimization requirements can be achieved via the parameters which can be tuned at run time. The system under control in this paper is a real-time collaborative system. The system allows users in various locations to collaborate in real-time via audio, voice and text and ensures that all users in a system view the same data at the same time. This section will focus on related work in such collaborative systems in order to determine the self-optimization requirements of such systems.

2.2.4.1 Apache Wave

One such collaborative system was Google Wave which used AJAX and HTML5 in order to allow users to collaborate from their browsers. The project has been closed by Google but open sourced as an Apache project [47]. The client side presents users with a list of waves which were recently updated and users can select which of these waves to look at. While looking at a wave, which is a collaboration between multiple users, the system ensures that all users see the same data. This means that changes done by one user will be reflected on the screens of all the other users in the same wave. The system also allows developers to create their own applications for the waves which can be synchronized. Figure 2.10 shows two browser windows sharing the same wave with a map and some annotations on the map.

On the server side the system uses a Java based server written on top of Jetty - which is itself a Java based application server. The server uses XMPP for federation of data in a cluster of servers. Each of the servers stores information regarding the waves running at a certain time on the given server. The wave information is

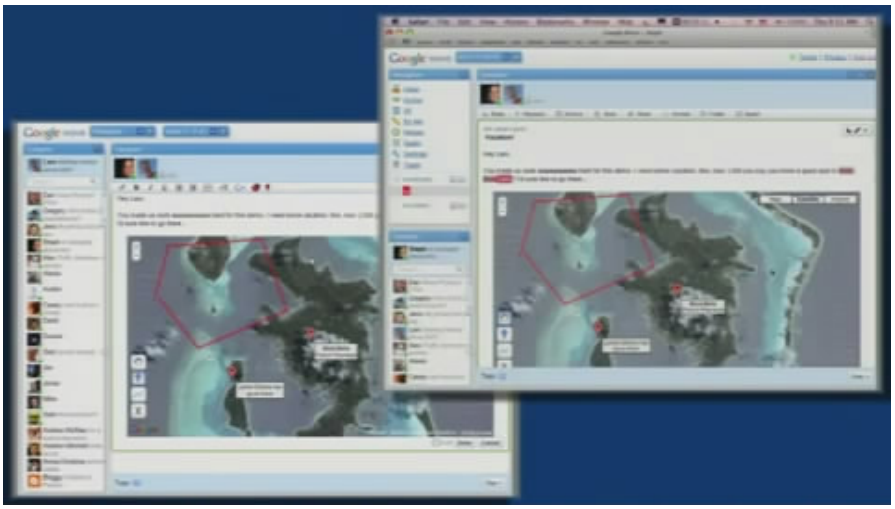

FIGURE 2.10: Collaborative Maps in Google Wave

then distributed using XMPP to any other servers which have users in the same wave. The originating server is responsible for hosting, processing and concurrency control of the waves running on it. Wave does not provide by itself any audio/video collaboration capability and as such would require an external server to provide the real-time streaming capability necessary for this type of collaboration.

Since the server side is not simply an application for an application server, self-optimization for such a system can not be simply provided by trying to control the Jetty engine in which the system runs. Furthermore if the system were to add audio/video capabilities two self-optimization control loops would be needed - one for waves and one for audio/video. Simply increasing the number of servers in the cluster as demand increases could be a solution for self-optimization but the relative slowness of XMPP messages due the serialization/deserialization of such messages makes it unclear how the client response time would be affected. Moving clients to the same server to share the same wave instance would decrease XMPP based serialization/deserialization problems and could be a self-optimization solution but such an approach might increase contention problems as large numbers of users attempt to modify the same wave at the same time.

2.2.4.2 Adobe Tour Tracker

Another real-time collaborative system was Adobe's Tour Tracker which was used in order to show the capabilities of Flash and Adobe Flash Media Server. The system was different from other real-time collaborative systems in that it did not

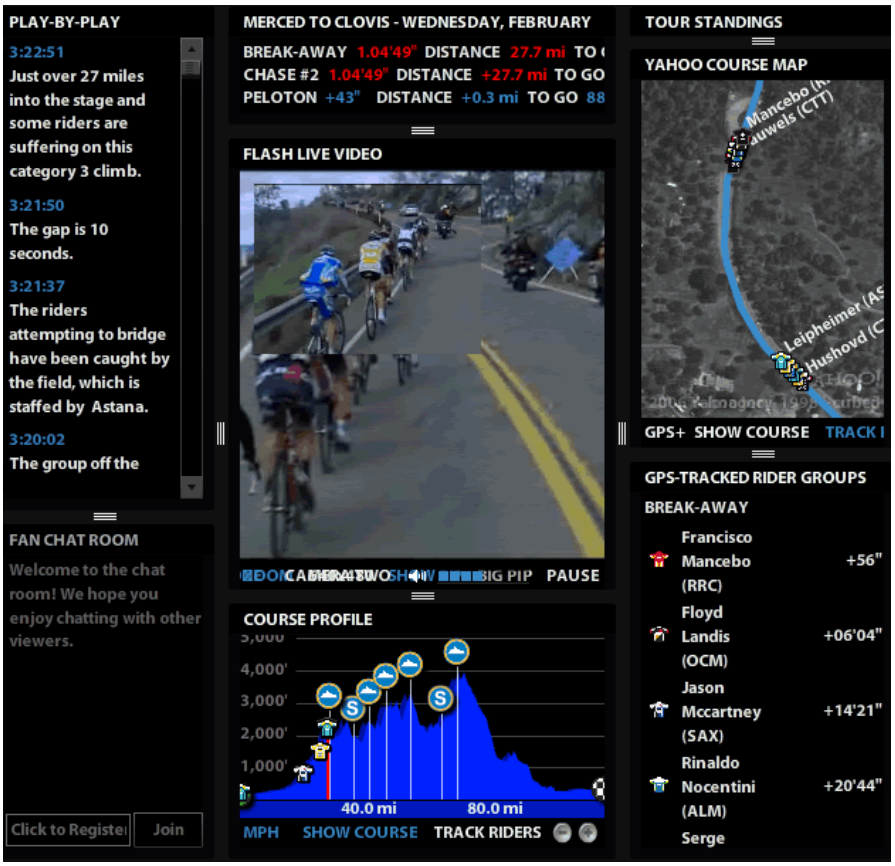

FIGURE 2.11: Adobe Tour Tracker Client View

allow multiple users to share the same view or to share data via audio/video/text collaboration. The goal of the system was to present to the clients of the system information from the Tour of California. The clients could see where various bikers were in the race using GPS data sent by the clients and could select to follow a certain biker and see a video showing the progress of that biker. Figure 2.11 shows the client view.

On the server side, the system used Adobe Flash Media Server which is Adobe's implementation of an RTMP server. RTMP itself is a protocol specification defined by Adobe which allows a Flash based client to communicate with a server and provides serialization/deserialization of data to/from the client. The server receives data from the various sources - video, GPS, photos, etc. and then sends it to the clients interested in such data. In order to provide capacity, clients were actually connecting to a cluster of servers which was selecting a server for the client to use randomly on connection. It is unclear how data was received by the servers from the various sources - whether it was sent to all servers in the cluster or multiplexed in the datacenter.

Due to the fact that the collaboration was so simple however, servers did not need to communicate with each other. As such self-optimization can simply be achieved by adding more servers to the cluster. A different solution for self-optimization if the cluster can not be scaled up more would be to lower the quality of the video/audio received by clients if bandwidth is an issue.

2.2.4.3 Browser Based Groupware

In [48] the authors study if only browser technologies like HTML5, AJAX and Comet can be sufficient to create groupware systems. The goal of the authors is to determine if simple browser technologies offer sufficient performance for real-time groupware systems while avoiding the use of any browser plug-ins. The author's work is important because it focuses on what kind of performance is important for real-time groupware applications. For this, the authors compare the networking capabilities and performance of three web-based networking approaches: repurposed technology for delivering web pages (Comet), network mechanisms inside the browser (Web Sockets) and plug-ins (Java Applets). The authors note the following important requirements for networking which real-time groupware applications have, depending on the type of groupware application:

1. Card and board games - generally do not have any large network requirements as only one user does an action at any point in time.
2. Chat rooms without audio/video - Low network requirements, but which increase as user numbers increase. If system supports text as you type appearing on all screens the number of messages increases and latency becomes more important.
3. Shared workspace - if such systems include the ability for all participants to see the mouse movements the rate of messages can be relatively high, but message size is quite low. Latency is also important.
4. Complex games - the rate of position update is quite high and messages can be relatively complex although usually developers use client side interpolation which can alleviate some network requirements. Latency quite important.

5. Videoconferencing - requires large bandwidth, which increase heavily with large numbers of users and also low latency, especially for audio.

The results of the authors are that browser based technologies can offer sufficient capability for groupware, as some of the methods offer performance similar to the Java Applet in terms of rate of sending and receiving messages to/from the server. One thing to note is that the authors do not test any kind of videoconferencing, which would be the most demanding on the network because browser based technologies did not support access to microphones or video cameras when the paper was written.

2.3 Related Work Conclusions

The goal of this thesis is to develop an autonomic system which optimizes a geographically distributed cloud based real-time collaborative application. Investigation into related work have lead to a number of conclusions.

- The best approach to create an architecture for such a system is to use a decentralized self-organizing architecture which scales well with large numbers of servers in the cloud.
- A number of approaches exist to develop autonomic systems, from machine learning to control theory based systems. Control theory offers the best ability to ensure guarantees such as stability and responsiveness and offers a number of existing algorithms which have been proven to work. This ensures a better chance of the system being put into practice in real-life as guarantees are offered to the system administrators.
- The system must be able to manage the response time and latency as seen by end-users which is very important in real-time collaborative applications.
- All current work into control theory for autonomic systems focuses on black-box models due to how complicated it is to develop a model from first principles for an IT system. This thesis proposal will present a system based on first principles.

- There is no work performed, as far as investigated, which develops a geographically distributed cloud for real-time collaborative applications. The closest to such a system is Apache Wave which federates the data in a session across multiple servers, however no geographic knowledge is used in the system. Such a system will be presented in this thesis proposal.

The requirements of the system and an initial control model are presented in Chapter 3.

Chapter 3

Architecture and High Level Design of Cloud Self-Organizing Servers

The work presented in the thesis relates to the design and implementation of a self-optimized cloud-based client-sever systems using self-organization algorithms. The algorithms developed will be tested on a cloud based solution for a geographically distributed collaborative application. This chapter describes the design of this application server. The chapter presents also the modifications needed for migrating the server for the collaborative application from a single server to a distributed server deployed on multiple clouds. Figure 3.1 shows the server architecture with a single server while figure 3.2 shows the server architecture under a distributed Software as a Service (SaaS) model.

The Software as a Service deployment model for the collaborative application has the following important characteristics:

1. Server-client application - the system is a client-server application in which clients connect to a server and messages sent between clients go through the server, which distributes the messages to the appropriate destination clients.

2. Cloud based application - clients connect to a cluster of servers which is virtualized, and clients communicating with each other could connect to different server instances in the cloud.

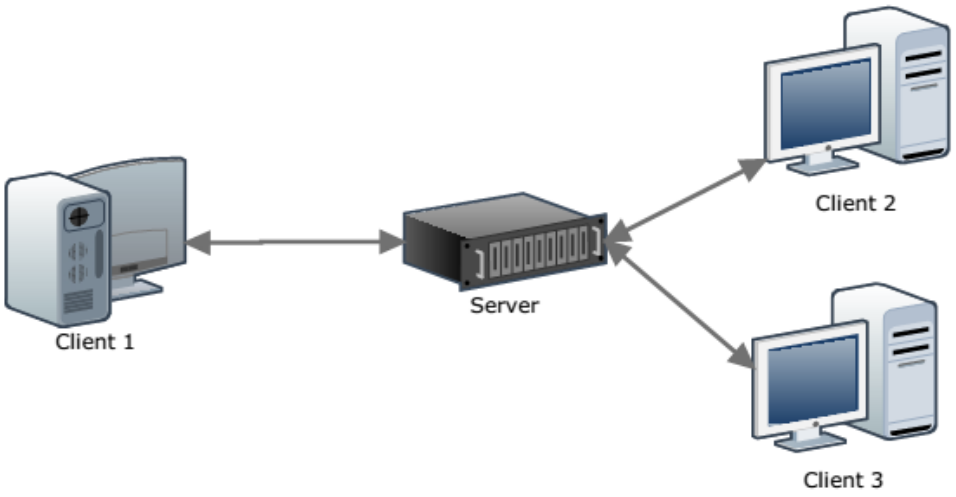

FIGURE 3.1: Single Server Deployment

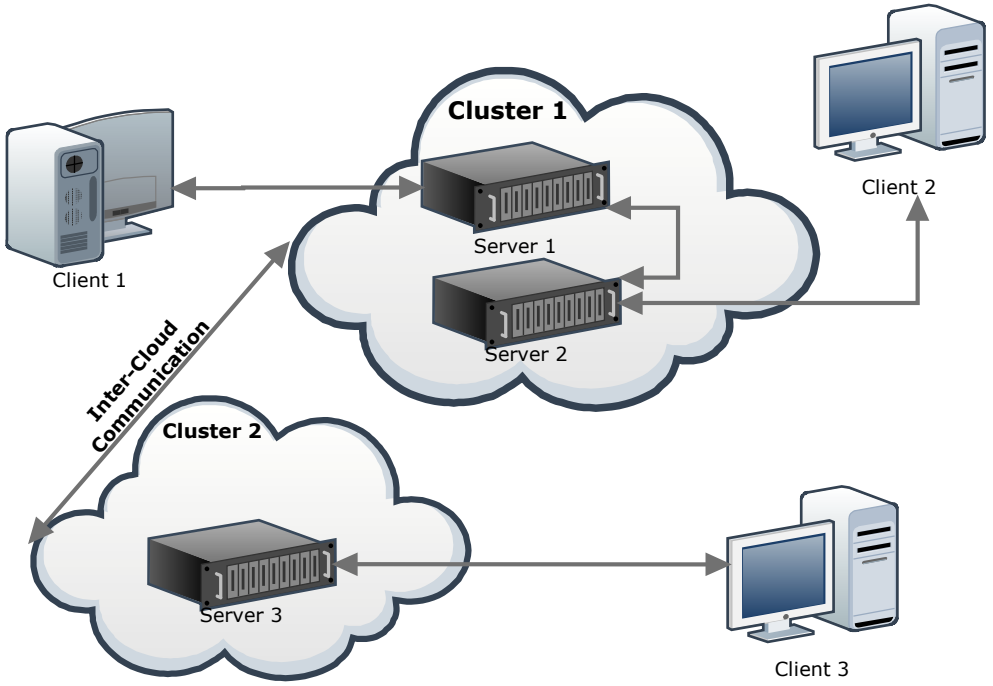

FIGURE 3.2: Software as a Service Deployment

3. Collaborative application - the system (servers in collaboration with the clients) ensures that clients in the same collaboration session view the same state of the shared "workspace".

4. Geographically distributed application - clients connect to one of multiple clouds which are distributed in various locations. Clients from different clouds can still communicate with each other.

5. Containerization - the servers are deployed in the cloud inside containers.

3.1 Requirements

The above characteristics of the application result in a number of functional requirements for the application.

Client Presence Clients can see when their contacts come online, go offline, join a collaborative session and become busy.

Client Synchronization Clients in the same collaborative session must see the exact same thing in the collaborative part of the application.

Client Communication Clients in the same session can communicate with each other via text, audio, video.

Session Size No restriction is set on how many clients in a session can broadcast video/audio at the same time.

Session Setup Any user can invite another available user to a collaborative session (pending restrictions created by the session's controller).

Session Number Users can only participate in one session at any time.

Session Control Either any user can choose what is viewed in the shared collaboration panel or the session's manager can restrict the control to himself only.

Session in Progress Synchronization Clients who join a session already in progress must be synchronized to the state of the session.

On top of the functional requirements, the system also has a number of non-functional requirements:

Client Connection Clients connect to one of multiple clouds based on a metric which defines how good the performance of the server is for the client.

Server Cluster Servers are combined in clusters, which from the point of view of the client look like a single server.

Server Cluster Scaling Server clusters can scale up and down dynamically.

Server Cluster Location Clusters are distributed geographically.

In Cluster Communication Servers can send messages to other servers in the same cluster.

Message Distribution Messages are distributed via a Group Membership System (GMS).

Out of Cluster Communication Servers can send messages to other servers in a different cluster.

Gateway Messages are distributed via a 'gateway' which sends messages to other clusters or receives messages from other clusters.

Containerization The servers and related infrastructure is deployed inside containers.

3.2 Single Server Architecture

The first part of this chapter will focus on the architecture of a single server, while the later part will show how the server was modified and what other components were added to the system in order to support the requirements for cloud and geographically distributed deployment including containerization. The client side uses a browser based technology in order to provide better user accessibility, while the server side was developed as a Java based application.

In order to achieve better modularity, the server is split into four core modules plus two other modules used for client side application communication. The four core services are:

UserStateService This service is used in order to indicate changes in user presence (coming online, going offline, currently busy), to retrieve the list of connected users and also to store the list of contacts for all connected users. The list of contacts is stored in order to decrease the number of messages related to users coming online. When a new user connects to the server, the server simply looks through the lists of already connected users to determine which users are contacts of the newly connected user. These users receive information that the new user is online, as well as the newly connected user receiving a list of contacts which are online.

SessionService This service manages the various collaborative sessions created by users. It stores a list of currently existing sessions, a map of clients to sessions as well as a map of clients which have been invited to a session but have not yet accepted/rejected the invite to rooms. Two other maps exist which are mirrors of each other, mapping invited clients to inviter clients and vice-versa. The service is used to create new sessions, join and leave sessions as well as broadcast messages in a session. The service is also responsible for synchronizing new users to the state of a session - however the server does not store the exact state of a session.

WebcamVideoStreamService This service is responsible for managing user's audio/video streams. It allows users to start/stop streaming and notifies the appropriate users that a new stream was started (using information from the *SessionService*). For receiving users, the service allows users to subscribe to streams published by other users.

ServerStatsService This service generates statistics regarding the server's performance metrics, including CPU usage and memory usage, as well as gathering stats from the previous three services like bandwidth used for streams, number of users connected, sessions active, etc.

On top of these four core services, two other services exist which support client side "applications".

DocumentService The document service allows clients to search for documents by keywords or by user who created the documents. Documents in this case include various types (spreadsheets, presentations, text files, etc.) which are uploaded by users to the server where they are converted to browser

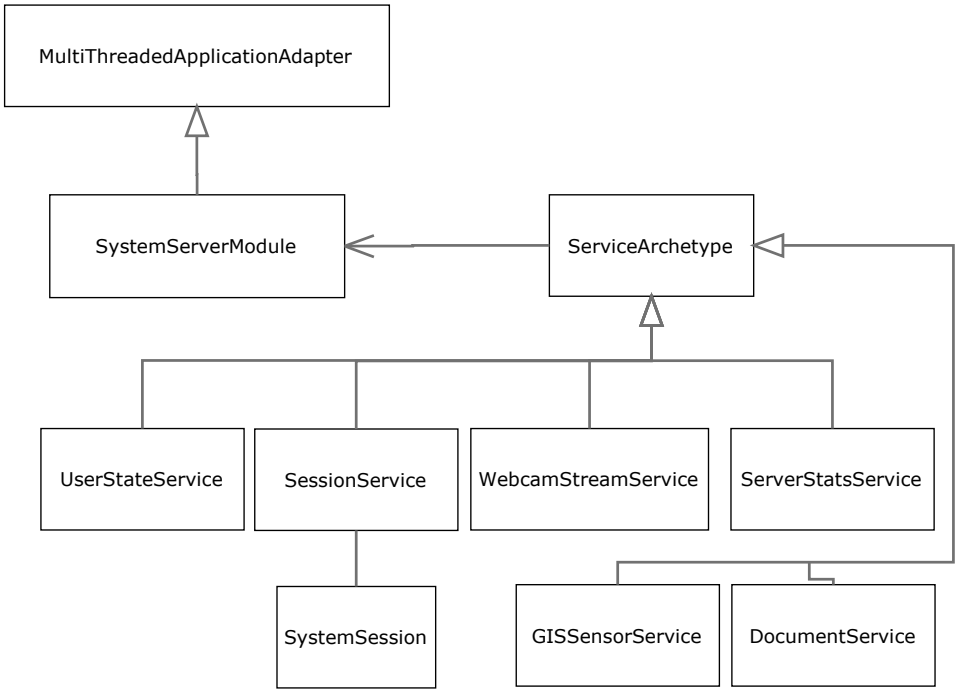

FIGURE 3.3: Server Side Services

viewable date for viewing in the collaborative area of the client application. Documents can be defined as public or private and the search will ensure correct visibility of documents.

GISSensorService The GIS sensor service allows users to receive live data from Geographic Information System (GIS) sensors. Users can discover sensors and subscribe and unsubscribe to/from sensors. Once subscribed to a sensor, the service will create the respective subscriber to GIS data and forward the data when it is available to the clients.

Outside these six services, the server also contains a main application module called *WatchTogetherServerModule*, and a *WatchTogetherSession* class which maintains server side information regarding the state of a single collaborative session including the user who is the host of the session, the session's unique ID and the clients in the session. Figure 3.3 shows the various classes involved in the server. All services extend a *ServiceArchetype* class which injects information about the main server application.

3.2.1 Client-Server Communication

In this chapter *User* refers to the person using the collaboration system while *Client* refers to the software running inside the browser which connects to the collaboration server. Numbering is consistent between client and user, for example *Client 1* is the client used by *User 1*.

The communication between client and server can be split into a number of types of messages: connection setup, session setup, session synchronization, audio/video communication.

Connection Setup

Figure 3.4 shows the connection setup process as a sequence diagram. The diagram shows three client connections: *Client 1*, *Client 2* and *Client 1 PrevConn*. Two of the connections *Client 1* and *Client 1 PrevConn* are connections associated with the same user (shows by some form of unique user ID). At the time when the diagram begins both *Client 1 PrevConn* and *Client 2* are actively connected to the server. Also the users represented by *Client 1* and by *Client 2* are contacts of each other.

The diagram begins with the server module receiving a new connection. The connect message contains a field specifying the unique ID of the user which is trying to connect. The server checks to determine if there is already an active connection for the specified user ID. If a connection already exists the previous client connected with the ID is sent a disconnect message and the connection information is purged from the server. The user ID is associated with the new client connection and the client is sent an accept connection message. Upon reception of accept connection, the client sends to the state service a message requiring the server to notify the user's contacts which are online that the new user has come online. The notifyOnline message passes a list of contactIDs to the server which represent user IDs which are contacts of the newly connected user. The state service goes through its list of connected users and determines which of these users are in the list received from *User 1*. Each of these clients receives a notification regarding the online status of *User 1*. At the same time, *User 1* receives notifications for each of contacts which are already online. Since the connection and online notification is done in two steps, it is guaranteed that users

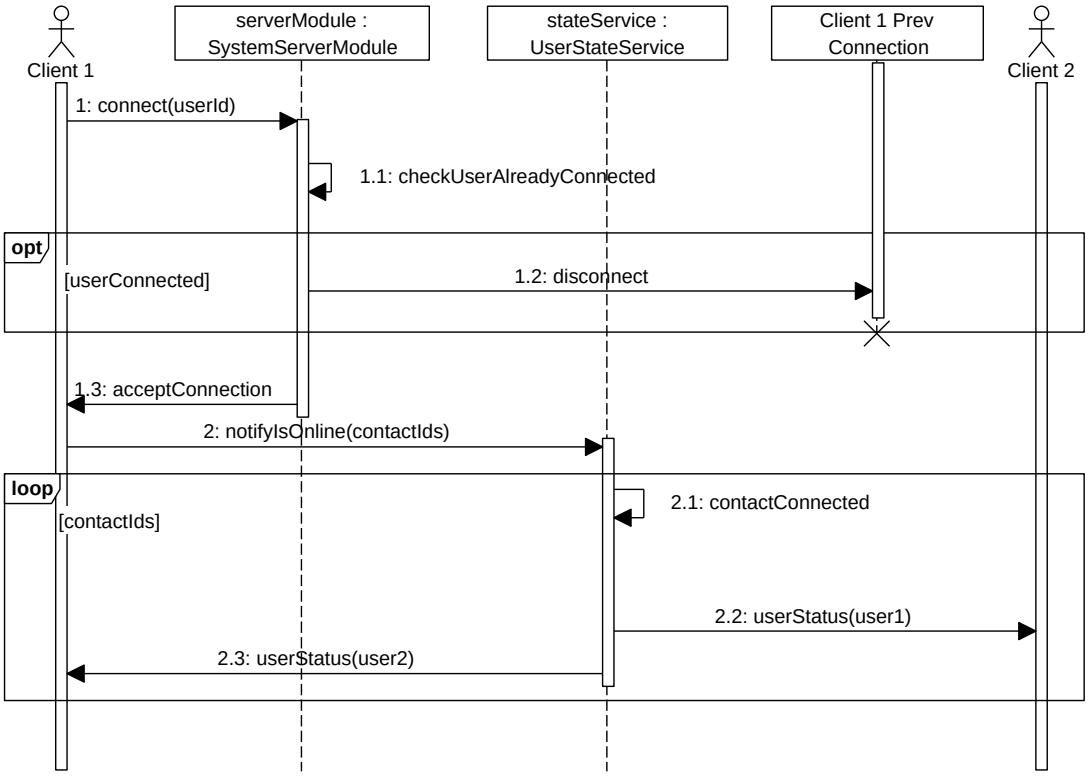

FIGURE 3.4: Client-Server Connection Setup

will know about each others status without need to employ semaphores. Even if two users connect at exactly the same time, by the time the notifyIsOnline message comes from either client the server will know that a user with the given ID is already connected.

Session Setup

Figure 3.5 shows the session setup process as a sequence diagram. The session setup process starts when a user (*User 1*) invites another user (*User 2*) to a collaborative session. The invite message is sent to the session service, which in turn checks the availability of *User 2*. The reason to check if the invited user is available at this time is due to the fact that a user could become busy by creating a session or accepting another invitation while the current invitation was in transit. As such *User 1* would be sending an invite thinking that *User 2* was still available. If *User 2* is still available, the session service checks if *User 1* is already in a session. If a session does not already exist for *User 1*, a new session is created and *User 1* is added to a session. The newly created session returns its unique ID back to the session service which stores it together with the actual session object. Once

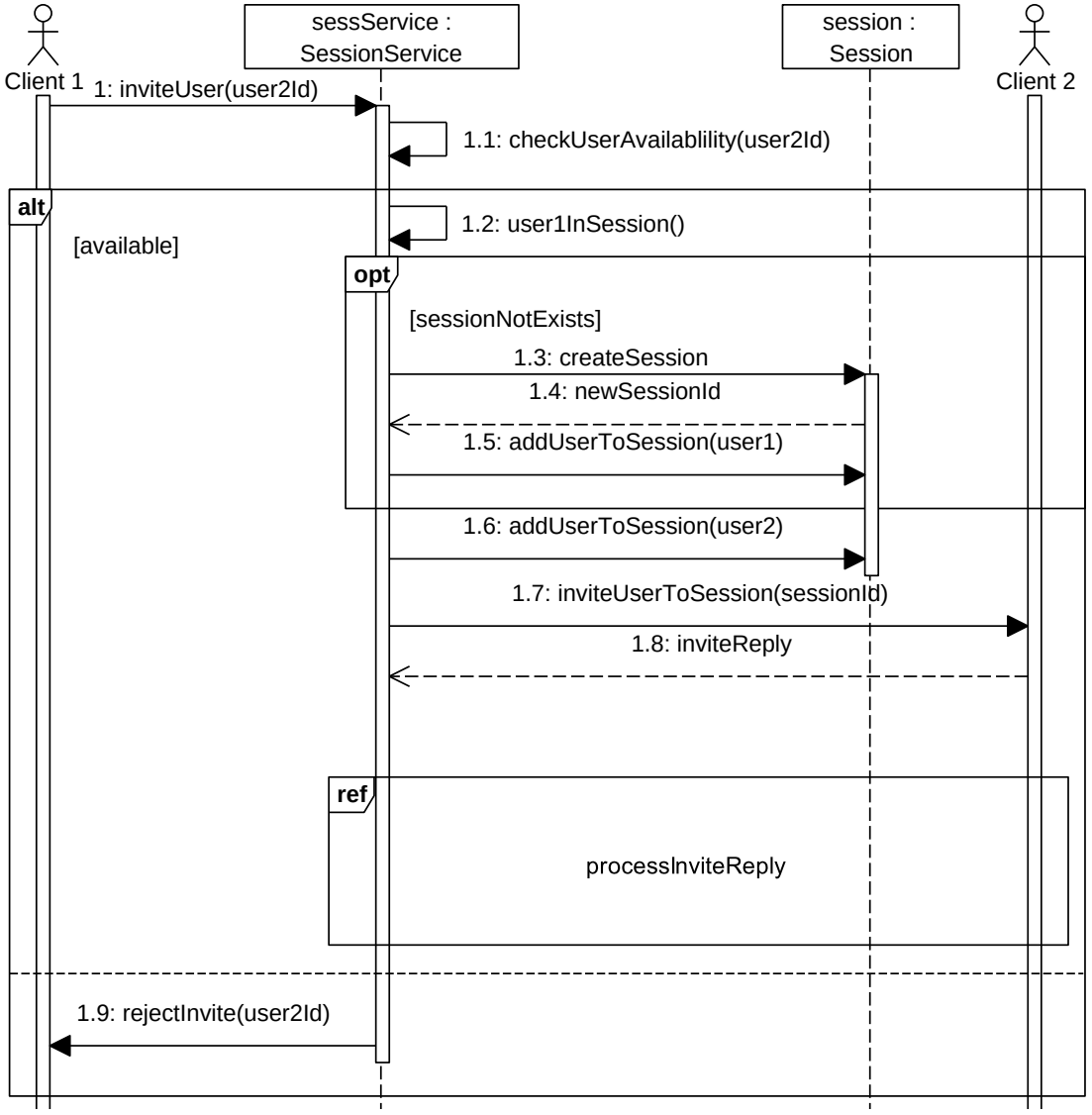

FIGURE 3.5: Client-Server Session Setup

the session is created a message is sent by the server to *User 2* notifying the user that the user has received an invitation. Once the user has decided whether to accept or decline the invitation (or the invitation timeout expires which results also in a decline reply from the client) the respective reply message is sent to the session service which processes it. The processing of the reply is shown in 3.6. The session service performs two extra actions which are not shown in the diagram: if either of the two users (invited or inviter) is available then the user's availability is changed to busy and all the user's contacts are notified about the status change.

Once the reply for an invitation is received by the session service, the service checks if it is an accept or decline reply message. If the message is an accept message the user is added to the session, which results in the client receiving a list of other

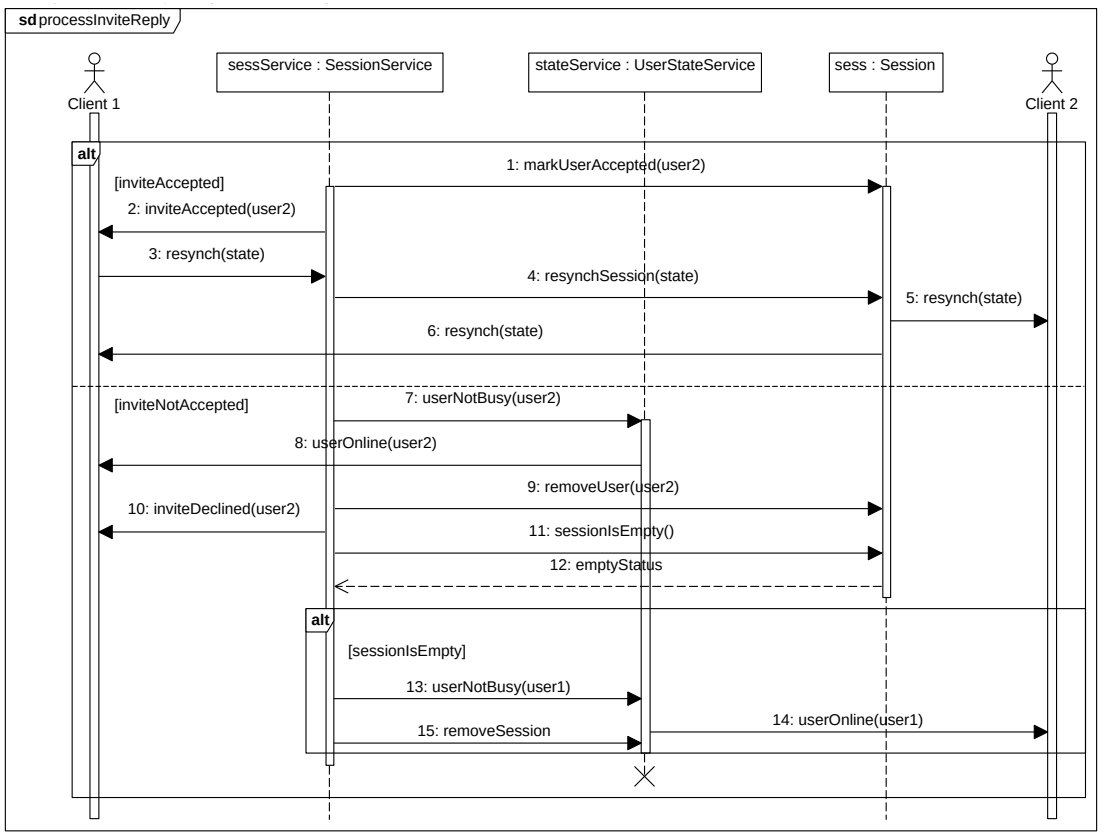

FIGURE 3.6: Client-Server Invitation Reply

users in the session and any users in the session receiving a message notifying that *User 2* has joined the session. *User 1* receives a message that the invitation was accepted. Finally, the session service asks the session to synchronize the new user to the session state. The initial synchronization mechanism is showed in figure 3.7. If the reply message is a decline message however, *User 2*'s availability is changed to show that the user is available to receive invitations and a message is broadcast to all the user's contacts to announce the status change. At the same time, *User 1* is sent an invitation declined message. The server then checks to see if the session is empty - a session is defined as empty if there is only one user in the session and there are no invitations pending reply. If the session is in fact empty the last user in the session has the status changed to available and the session is destroyed.

After a synchronization is requested, caused by a new user joining a session, the session service obtains a synchronization lock on the session and asks the session to synchronize the new user. The session finds the client which is the session's host - the host will be either the client which initiated the session by inviting another user or another user assigned randomly in case the initial host leaves the session. The new user which is to be synchronized to the session is then added to

a list of users waiting for synchronization messages. If the list of users waiting for synchronization only contains one user a message requesting the synchronized state of the session is sent to the session host client, which replies with the current state of the collaborative session. The synchronization lock is released at this point. Upon reception of the current state by the session service, the service obtains a synchronization lock on the session and passes the synchronization information to the session, which in turns broadcasts the synchronization state to all the clients in the list of clients waiting for synchronization messages. Once all the messages have been sent, the lock is released. The reason to use a list of clients waiting for synchronization messages is in order to ensure that the host does not receive a large number of messages requesting synchronization. Imagine a user inviting a large number of contacts to a session, and a large number of these contacts accepting nearly simultaneously. The first accept message will generate a synchronization request, however all other accept messages received by the server before the host replies with the synch state will simply wait and not generate extra synchronization requests. Once the synch state reply reaches the server, all users waiting receive the same state. At the same time, the reason to use a semaphore for access to the two synchronization methods in the session is in order to ensure consistency in behavior. If for example, an accept reply is received while the server is in the process of broadcasting the synch state to the clients already waiting, it is better to have the new client wait until all the broadcasts have finished and then ask the host for new synch data instead of trying to add the newly arrived client in the broadcast group for the message.

Session Synchronization

The process of maintaining synchronization between users in the same session is relatively easy from the server's point of view, as the server only acts as a relay to broadcast the message to all the users in the originating user's session. The server does not do any processing of the message and never examines the contents of such a message. Figure 3.8 shows a sequence diagram of the mechanism used for session synchronization. Initially, the originating client sends a message to be sent to all the users in the same session. Such a message is generated by the user performing some form of action in the web based client which should be reproduced on all clients. Upon reception of the request, the session service determines the session which the user is participating in and then broadcasts the respective message to

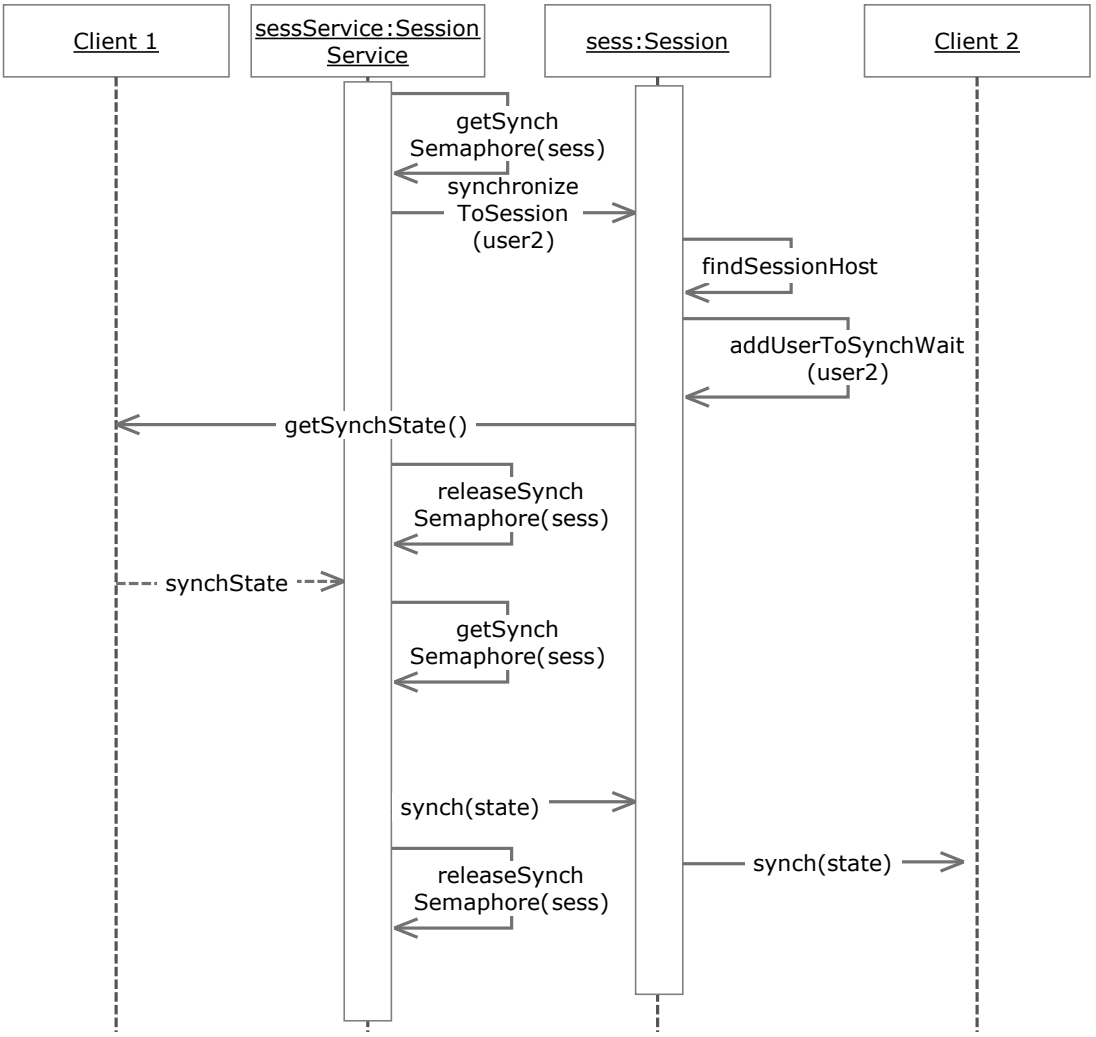

FIGURE 3.7: Client-Server Session Initial Synchronization

all other clients in the session. Note that no semaphores or locks are used to block access to broadcast messages. This is due to the fact that even if there are for example two users sending synchronization messages to the server at the same time, it is impossible for the server to know which was the first message due to network delays.

Audio/Video Communication

The audio/video communication is performed through an asynchronous setup. Upon reception of a new stream, the server can not simply start sending the stream to the other clients in the session. It is the responsibility of the clients to request that the server starts sending a certain stream. Because of this, the server

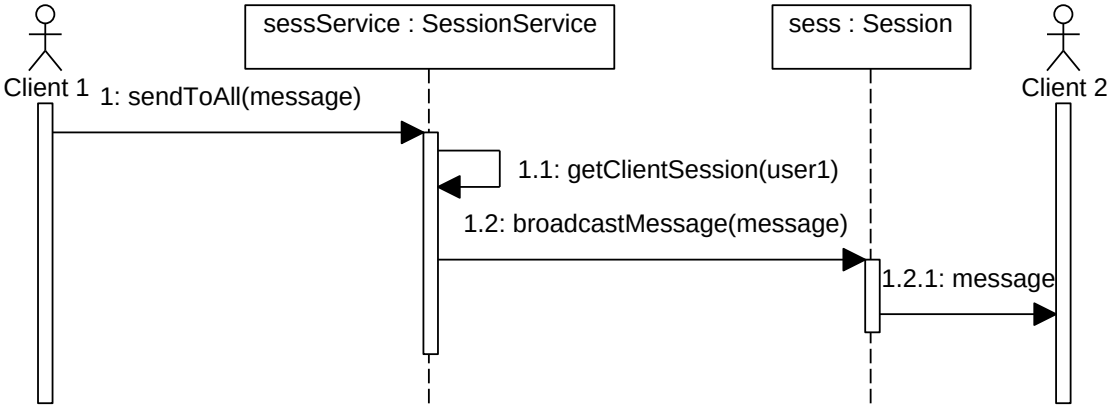

FIGURE 3.8: Client-Server Session Synchronization

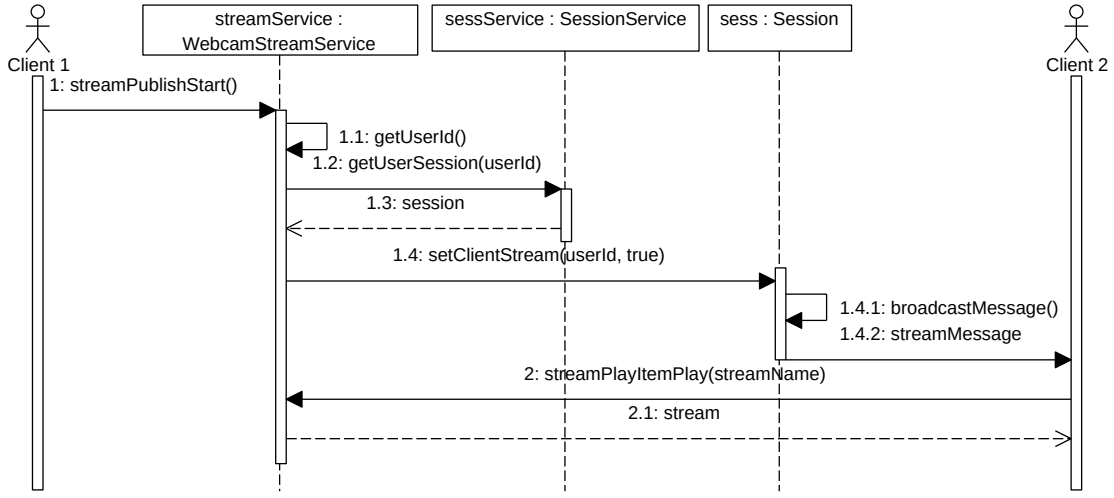

FIGURE 3.9: Client-Server Stream Start

notifies the clients that a new stream has started and then the clients request that the server sends the stream.

The stream start and stream stop setups are very similar. The process starts with the client starting or stopping a stream which generates a streamPublishStart or streamBroadcastClose event respectively. Upon reception of the message, the webcam stream service retrieves the client's ID from the stream metadata and then asks the session service to mark the respective user's stream as started or stopped. The session service finds the user's active session and sets the user's streaming information in the session as either started or stopped. The session then generates a message to be sent to the other user's in the session and broadcasts the message. If the message received by the clients is a stream started message, the client asks the server to start sending the stream which generates a streamPlayItemPlay event and the server starts streaming the data to the client. If the message received by

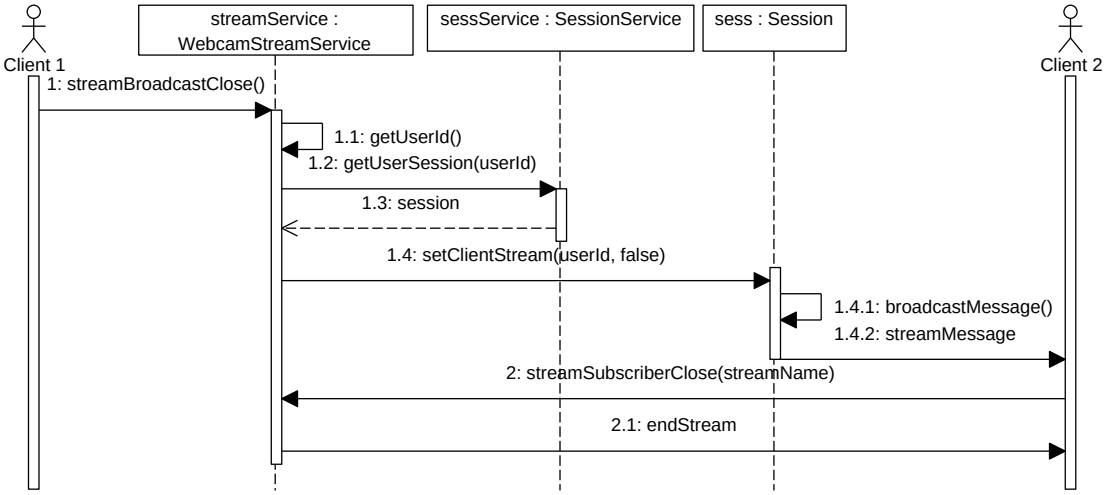

FIGURE 3.10: Client-Server Stream Stop

the clients is a stream stopped message however, the client asks the server to stop sending the stream which generates a streamSubscriberClose event and the server stops streaming the data to the client. The webcam service also stores the information regarding which clients are streaming at any point in time. This is done in order to let any client which joins a session know if it should subscribe to streams from the users which are already streaming their video.

3.3 Clustered Server Architecture

In order to achieve the goal of having the server capable of being deployed as a cluster in which users connect to any server and can still communicate with any other user even if that user connected to a different server in the cluster the single server architecture had to be changed. First of all, the clustered system required a method for the servers to discover each other and communicate with each other. Second of all, the clustered system required a way to replicate user information across the cluster. Finally, the clustered system required a way to proxy the webcam stream from one server to another such that users can receive the stream from the server they are connected to. This section will show how the system was modified to accomplish the above requirements.

3.3.1 Server To Server Communication

Since the server cluster is not static in size and is allowed to scale up and down based on demand a very important ability of the servers is to discover new servers when the servers start and for servers to broadcast when a server leaves the cluster. In order to achieve this goal, the cluster architecture uses a Group Membership System (GMS) to create a group which the servers join and leave. The servers then use the group created by the GMS 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 collocated in the same network and not interfere with each other's inter-server communication. The GMS used for this is the JGroups system [49], which is the same system used by the JBoss Application Server for clustering. More specifically, the architecture adds an extra component in the form of a Gossip Router which runs on a specific host and port. The Gossip Router is responsible for registering and deregistering clients from groups, as well as for responding to queries for the members of a group. In order to deal with group members crashing, members send periodically a refresh message to update the status of their group membership. If a long enough period passes without a refresh from a group member, than 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. One issue with the Gossip Router is that it can be seen a single point of failure in the architecture. Currently the cluster assumes that the Gossip Router can not fail, however if redundancy is needed the system could use a backup Gossip Router in case the main router fails.

Each server upon start-up creates a channel through which it communicates with the other servers in the cluster. Communication can be either point-to-point between two servers - if the sender of the message specifies a destination address - or a broadcast inside a group - if no destination address is provided. After a server creates a channel for group communication, the server connects to a predefined group based on name and then broadcasts in the group the servers information including host address, port on which the Red5 server is running as well as the name of the application. These three values are used to uniquely identify a server application. By depending on all three values, the cluster allows mixed systems in which either a host runs more than one Red5 server - each on a different port, or even a single Red5 server runs more than one application which joins the group

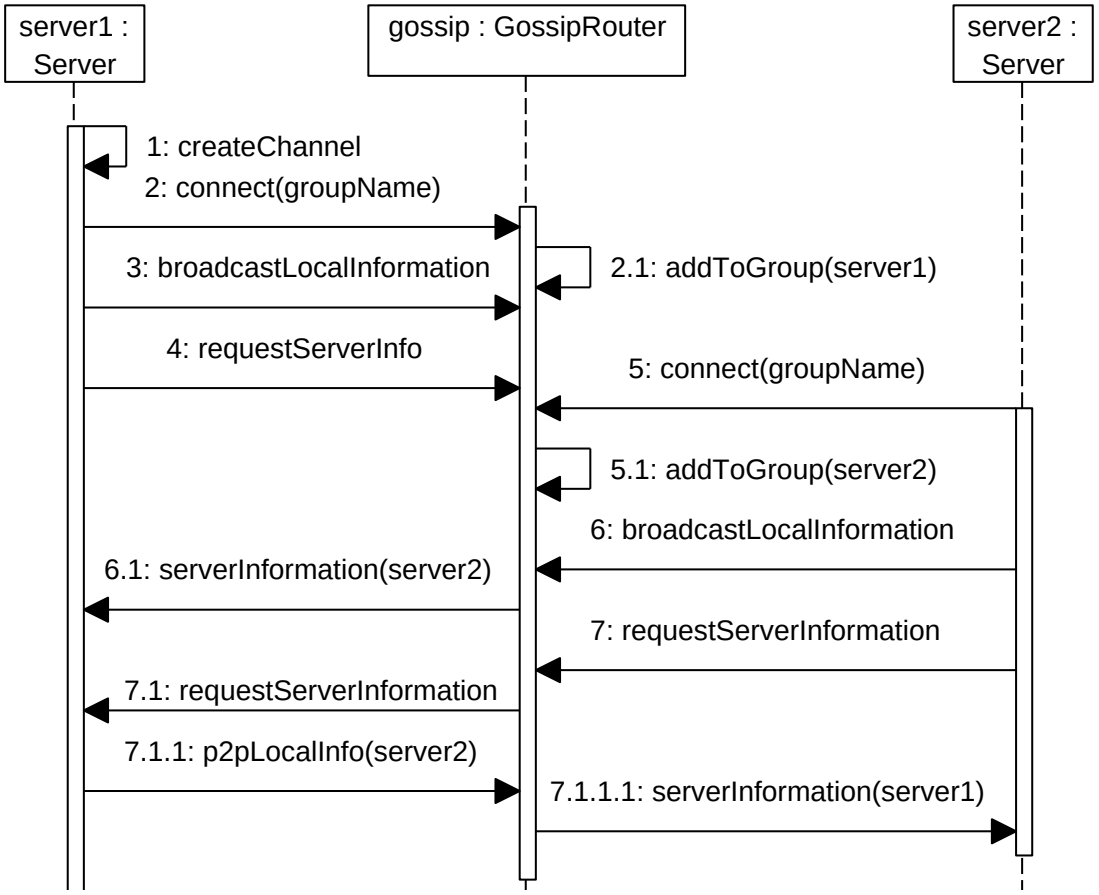

FIGURE 3.11: Server GMS Join

- in which case both the host and port will match for both applications. Figure 3.11 shows the process of servers joining a group.

Note that after a server connects to the group, it broadcasts its local information to the group. If other servers already exist in the group - in figure 3.11 when *Server 2* joins - the gossip server broadcasts the message to all the other servers. Upon reception of a new server information message, if the message does not have a specific destination which would mean it was a broadcast message and not a point-to-point message, the receiving server sends a reply with its own information in order for the newly joined server to know the other servers in the cluster. This reply message is sent as a point-to-point message in order to prevent the newly joined server from also replying to a new server information message which would lead to an infinite loop.

In order to allow for Inter-Server communication, two components were added to the server architecture - a group manager and a group receiver adapter. The group manager creates the communication channel and performs the group logic of the

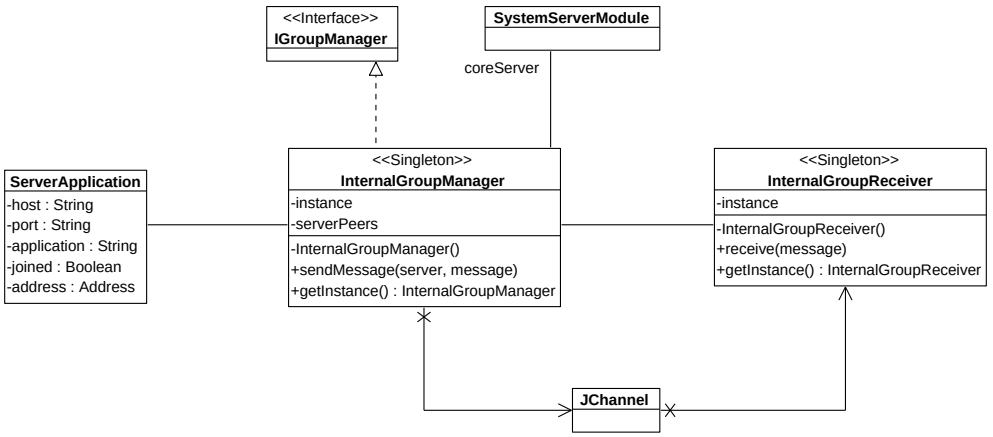

FIGURE 3.12: Server GMS Classes

server, the group receiver adapter receives messages from other servers, parses them and passes the message to the corresponding method in the group manager. On top of these two components, the group system also uses a number of messages for passing data between servers. For the moment, the only important message is the *Server* message. The other messages will be explained in the following two sections. Figure 3.12 shows the added components.

The server module creates a group manager on start-up, which sets up the JChannel and also creates the receiver. After creation of the receiver, the manager connects to the group and broadcasts a Server message which contains the local information which identifies the server instance (host, port, application) as well as a boolean which is true if the server is joining the group and false if the server is leaving the group. The group manager also stores a list of server peers which are Server objects representing the other servers in the cluster. New messages sent in the cluster originate from the group manager, while all messages from the cluster are received in the group receiver.

3.3.2 Cross-Server User Information Replication

In order for the cluster to allow clients to connect to any of the servers and still be able to communicate with each other certain user information has to be replicated across all the servers. For example, servers have to know to which servers other clients are connected in order to be able to route messages destined for those clients. In order to decrease the number of messages sent between servers, the status of the clients is also stored on all servers, even if the clients are connected

to other servers. As such, the user information replication is formed from two parts: data that is being replicated and messages used to replicate the data and keep it in synch across the servers.

Replicated Data

In order to encapsulate the knowledge of whether a client is local or remote, a class which represents a group client was added. This class encapsulates either a Red5 client, in which case the client is connected to the server containing this object, or a Server object, in which case the client is connected to a different server represented by the Server object. The group client also contains a send message method which is agnostic to where the client is connected. This method then delegates the actual message sending depending on where the client is connected to. Because of this, servers store a list of all group clients and not just local clients to the server. Furthermore, all servers store information regarding the activity status of all users (online, busy), not just local users. The reason to replicate the user status across all servers is in order to decrease server-to-server messages. For example, a new user connects to a server and sends to the server the list of contacts. The server then broadcasts a message to all other servers that a new user is online. The server however, does not need to wait for its peers to reply to the connect message - and in fact no reply is expected - before notifying the newly connected client which of its contacts are online and which are busy. Figure 3.13 shows the static view of the group client by extending figure 3.12. Each server has a list of *GroupClients*, where each group client has either a localClient object or a remoteServer object. If both objects are set (which should not happen in practice) the client is assumed to be local and the localClient object is used for communication. The figure also shows the *ClientConnect* message which is broadcast by a server in a group whenever a new client connects. The message contains the user's unique ID, the server object where the client has connected and a boolean representing if the user has connected or disconnected.

Figure 3.14 shows the way the connection sequence diagram changes in a cluster. It is an extension of 3.4. The figure does not show the process for disconnecting a client which was already connected using the same unique user ID since the process is the same as the one in 3.4. As soon as a server - either the one to which the new client connects or one of its peers which receives a client connect broadcast

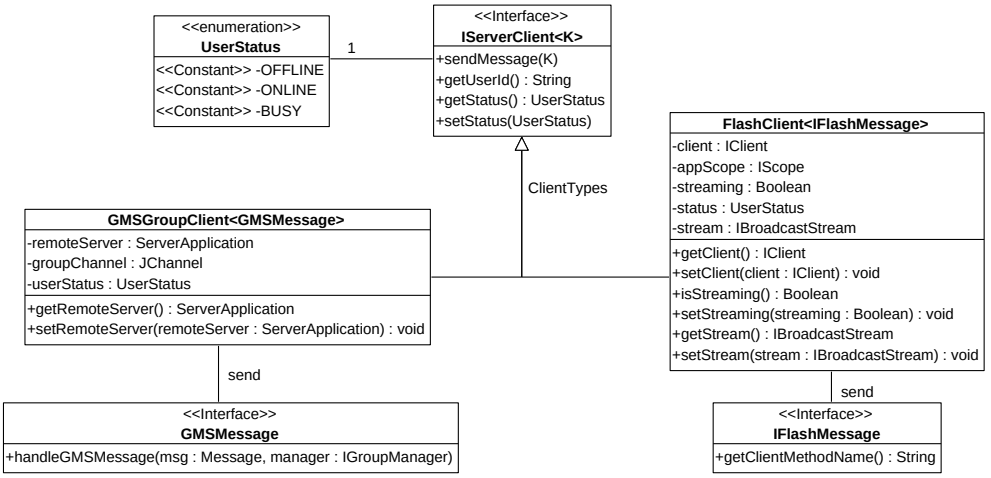

FIGURE 3.13: Server GMS Group Client Classes

- determines that a client with the same unique user ID is connected, it sends to that client a message notifying the client that it was disconnected. Upon receiving a new connection from a client, the server asks the group manager to broadcast a client connect message with the new user ID and at the same time accepts the user connection. The group manager creates a new *ClientConnect* message with the joined boolean set to true and uses the group channel to broadcast the message. The message is received by *Server 2* in the group receiver which determines if the message was a connect or disconnect message based on the value of the *joined* boolean in the message. Since this was a join message, the receiver passes the client information to the group manager which stores the client information. At the same time, a second client connects to *Server 2* and the process is repeated starting from *Server 2*. Upon reception of the notifyOnline message from the client, *Server 2* already knows that *User 1* is online and connected to *Server 1*, so it sends a message informing *User 2* that *User 1* is online. On the other side of the server-to-server communication, *Server 1* receives the message that *User 2* has just come online and since it knows the list of contacts for *User 1* it informs *User 1* that *User 2* is online. One thing to note, is that in order to simplify this diagram, the server entity contains both the actual server module and the user state service.

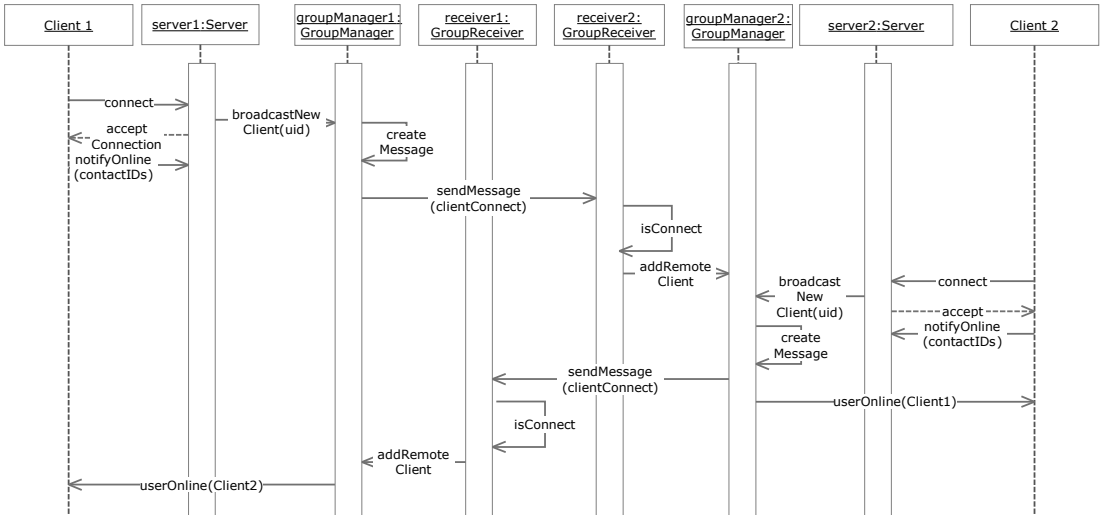

FIGURE 3.14: Server Group Connection Setup

Replication Messages

A number of messages are used in order to keep session data synchronized across the servers and clients connected to different servers. Figure 3.15 shows the various possible messages used for maintaining the state across the servers. On top of the *ClientConnect* message which was described previously, the system uses three types of messages and an abstract class. The abstract class named simply *Message* defines three properties that all messages will have - clientID which is the unique user ID of the destination user for the message, method which represents the method which will be called in the destination client and params which is an array of parameters to be passed to the method. The first type of message used is *InviteMessage* which is used to pass invitation requests and invitation reply between users which are not on the same server. Invitation messages include in the message the session ID which the message refers to, as well as a message type variable which can be one of *invite*, *accept* and *deny*. For invite messages the params structure will contain the inviter's unique ID, while for accept and deny the params structure will contain the ID of the user sending the reply. The *SessionStateMessage* propagates session synchronization messages and simply contains a session ID pointing to the session which the message refers to. All other information is encapsulated in the method and params structures of the *Message* superclass. Finally, the *UserStatusMessage* simply extends the *Message* class without adding any extra property. The reason to add this message type is in order to be able to parse in the group receiver what kind of message has been received based on message class only. *UserStatusMessages* have to update the user

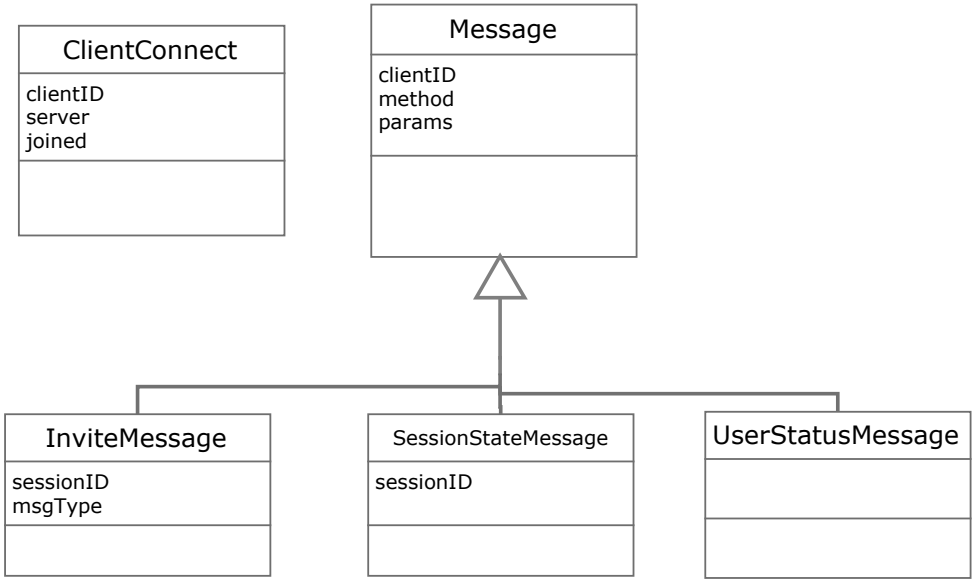

FIGURE 3.15: Server GMS Group Messages

status in remote servers as well and as such it is not sufficient to simply forward them to the destination client like some of the other *Message* implementations which are not processed by the server.

The processing of group replication messages at the receiver is relatively simple. Upon reception of a message, the group receiver determines the message type. If the message is a *ClientConnect* message then the server uses the process previously described.

1. If the message is not a session state message, then the message is sent to the destination client and then the server process the message to update its own internal state.

2. If the message is a user status message, then the remote user's status is updated in the server.

3. If the message is an invite message, then the server creates a session if none already exists with the specified ID and performs the actions previously presented for the single server case.

4. If the message is an invite accept message, then the server synchronizes the new client to the session. The server will first try to use a local user to synchronize even if the local user is not the session host. If no local users exist in the session, then the server requests that the session host is used to synchronize the new user through a *SessionStateMessage*.

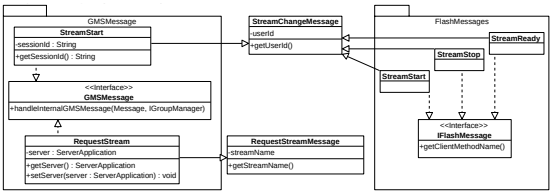

FIGURE 3.16: Server GMS Group Stream Messages

5. If the message is an invite decline message, then the server determines if the session is empty (in this case an empty session is defined as one with no local clients, or one local client and no remote clients). If the session is empty the server destroys the session.

6. If the message is a *SessionStateMessage* the server broadcasts it to all the clients in the session (including remote clients). The server takes care not to send the same message multiple times to the same server, if there is more than one client in a session on a remote server. An exception to broadcasting the message is done in case the message is a reply to a synchronization request, in which case only the clients waiting for synch and servers with clients waiting for synch will receive the message.

3.3.3 Webcam Stream Proxy

A very important consideration is the capability of the cluster architecture to proxy streams such that a client connecting to Server 1 can send a stream to a client connected to Server 2. It is also very important that if for example two clients connect to Server 2, only one stream is sent between Server 1 and Server 2. Because of these requirements audio/video streaming uses two types of messages: *StreamMessage* and *StreamProxyMessage*. Figure 3.16 shows the stream messages. *StreamMessage* contains the client ID which the message refers to and the type of stream messages which can be one of stream started or stream stopped. This message is generated when a user starts or stops streaming in order to notify other clients that a new stream has started, such that clients can subscribe to the stream. *StreamProxyMessages* are used to communicate between servers in order to request the creation of proxies. A *StreamProxyMessage* contains a server object which represents the server receiving the proxy stream and a stream name which is the name of the stream which should be proxied. Stream names always use the unique user ID of the streamer to simplify stream subscriptions.

Figure 3.17 shows the sequence used in order to set up a stream between two clients connected to two different servers including the creation of the proxy. This sequence diagram is an extension of figure 3.9. In order to simplify the diagram note that the *Server* entity contains both the server module and the stream service components while the *GroupControl* entity contains both the group manager and the group receiver. The diagram also skips some of the steps which are already shown in figure 3.9 such as obtaining the user ID from the stream publish request, and modifying the session state with regard to the user streaming. The broadcastMessage in figure 3.9 which was done locally, becomes a group broadcast which creates a *StreamMessage* containing the user ID and the type of stream message which is start in this case. Upon reception of the stream message, *Server 2* finds the session which *User 1* is part of, if any, and notifies the other local clients in the session that a new stream was started. Upon reception of this message, like in the local version, clients attempt to start playing the stream. Once the play request is received by the server, the server checks if it is already receiving a stream with the given user ID. If the server is receiving the stream already - either because the streaming user is also local or because a streaming proxy was already created - the server sends the stream data to the client. If the server is not already receiving the required stream, the server checks to determine if it has already requested the proxy from the other server. If the proxy request was already sent, the client is added to a list of clients waiting for the respective stream. If however, no stream proxy request has already been sent, the server asks the group manager to request from its server peer a proxy creation. The group manager creates a *StreamProxyMessage* with itself as the server and the ID of the user as the stream name and sends the message to the server to which the streaming client is connected. Upon reception of the *StreamProxyMessage*, the server creates a new *StreamProxy* with the correct destination and source stream and starts the proxy. The stream destination server simply passes the data to any clients waiting to play the stream.

The stop stream sequence is similar, with the difference that no stream proxy message is sent to the streamer client's server. Since the server detects when a client stops streaming, the server automatically destroys the stream proxy which also stops the stream and cleans up the stream resources used by the server. Upon detection of a proxy stream being closed, the server which received the proxy also cleans up any resources used for the stream. Figure 3.18 shows the tear-down sequence for a stream.

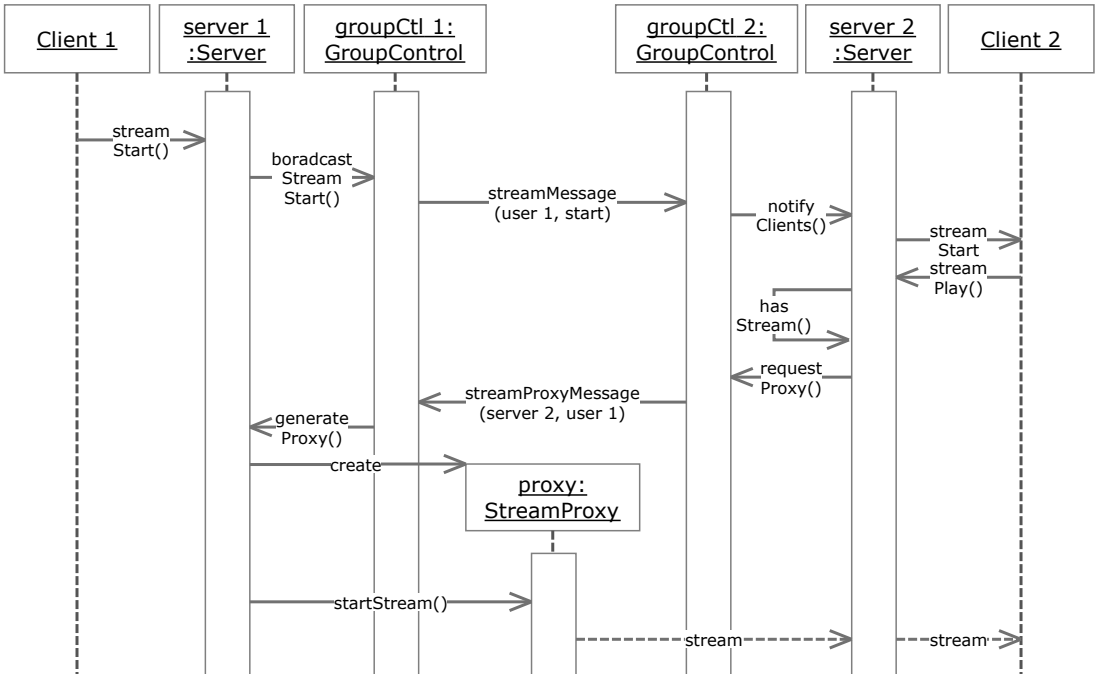

FIGURE 3.17: Server GMS Stream Proxy Setup

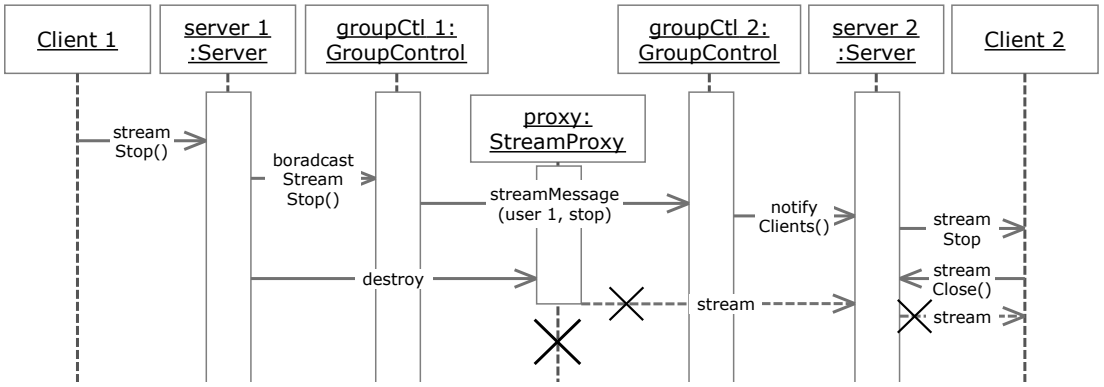

FIGURE 3.18: Server GMS Stream Proxy Teardown

3.4 Geographically Distributed Cluster Based Architecture

At this point, the thesis has introduced an architecture for a cluster of servers which are used to allow users to collaborate in real-time. Compared to the requirements presented at the beginning of this chapter, the architecture is still missing the geographically distributed nature. Two components are necessary in order to achieve the desired characteristics: a way for servers which are in different clusters to communicate with each other and an admission control system which redirects clients to the appropriate cluster based on some form of metric.

3.4.1 Cluster To Cluster Communication

While the server to server communication uses lists made of the servers in the cluster in order to know where the server peers are located, such an approach would be prohibitive for the entire distributed nature of the architecture. At the same time servers inside the same cluster 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 cluster can not know the host names or IP addresses of servers in other clusters, or even if they know them can not use the values. Because of this, a tradeoff has to be made between decreasing the number of messages sent between clusters and the global system state stored at each cluster. To achieve the cluster to cluster communication, the architecture uses a gateway component placed in each cluster (the section will also discuss the possibility of using multiple gateways). The gateway itself will be a peer in the GMS group created for the cluster, but instead of processing the messages like the servers do, the gateway will simply broadcast the message to the other clusters. In the case of client-to-client messages, if the destination client is connected to the local cluster (the gateway also stores a list of connected clients) then the message is not sent to other clusters. The gateway will also be responsible for receiving messages from other gateways, and broadcasting the messages inside the cluster. A special gateway will also be created for creating stream proxies between clusters. The messages which must be broadcast by the gateway are the following:

1. Client connected message - must be sent to all the clusters forming the system since clusters do not know ahead of time where contacts of the newly joined user are located.

2. Client disconnected message - must be sent to all the clusters forming the system since clusters do not know ahead of time where contacts of the leaving user are located.

3. Invite message - if the invited client is in the local cluster, then no action is taken; if the invited client is not in the local cluster, the message is sent to all other clusters.

4. Invite accept/reject message - if the inviter client is in the local cluster, then no action is taken; if the inviter client is not in the local cluster, the message is sent to the originating clusters (a list is kept for invites awaiting reply).

5. Session state message - the gateway stores for each session ID a list of clusters participating in the session (this is determined by checking the invite accept replies received from other clusters or sent to other clusters) and sends the message to the participating clusters. This implies that cleanup also has to be done by the gateway when sessions are destroyed or when another cluster no longer has clients participating in a session - a simple counter can be used to achieve this.

6. User status message - must be sent to all the clusters forming the system since clusters do not know ahead of time where contacts of the user whose status changed are located.

7. Stream message - must be sent to all the clusters forming the system since clusters do not know ahead of time where contacts of the user who started or stopped streaming are located. In the case of stream start messages, gateways store the originating cluster address in order to be able to route proxy messages. This data should not be stored the entire time that a user is streaming at all clusters. Because of this a new message must be added which notifies the gateway that a server is interested in the stream. If all servers reply with not interested, the data can be deleted from the gateway.

8. Stream proxy message - must be sent back to the originating cluster, based on the knowledge from the stream message received previously. Since servers from separate clusters can not communicate directly, a stream proxy gateway is used which allows a stream to be proxied between two clusters. Figure 3.19 shows the machines through which the stream is going.

In terms of static structure the gateway will be very similar to the group mechanisms used for each of the servers. The gateway will have a *GroupManager* and a *GroupReceiverAdapter*, the *GroupManager* will create the communication channel and join the group using a special ID in order for the servers to know the gateway and not assume it is just another server. The *GroupReceiverAdapter* will receive receive group messages and pass them to appropriate methods in the *GroupManager*. On top of these two classes the gateway will also use a *CloudManager* class which is responsible for communicating with the other gateways. Since the cloud structure can be assumed to be nearly static - clouds will not be created on the fly - the *CloudManager* will simply use a list of static IP addresses to know the location of the other gateways. In the case that a new cloud is added to the system, the IP

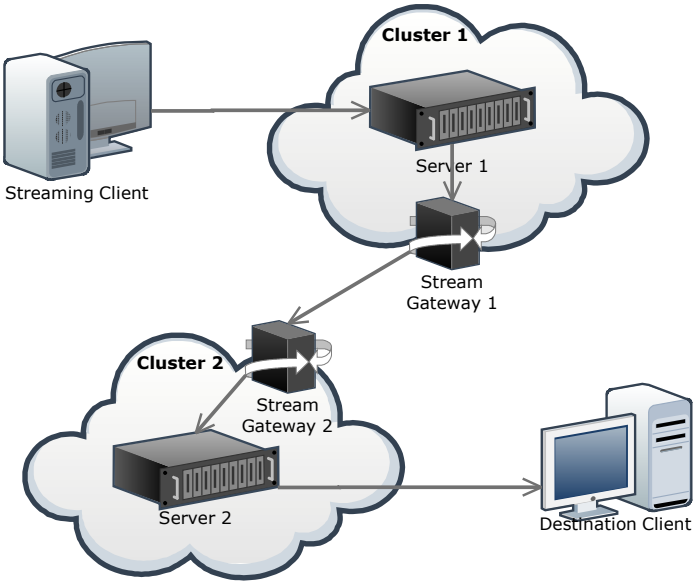

FIGURE 3.19: Geographic Cloud Streaming

address of the new gateway can simply be added to all the other gateways. This constraint also means that the gateways will represent single points of failure. The gateway system will use the same message classes used by the inter-server group communication system.

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 would also decrease the stress put on the gateways. Instead of using a deployment like figure 3.20, the system would use a deployment like figure 3.21. Stream gateways would use the same approach, with one gateway per communication link. If multiple gateways are used, the gateways in the same cloud could create their own group to quickly exchange messages without sending the messages to the servers.

The connect and disconnect message sequences are very simple as the gateway simply broadcasts the messages to the other clouds. Figure 3.22 shows the process used to setup a session across two clouds. The diagram assumes that an invite was sent by a client who is connected to *Server 1* in *Cluster 1* to *Client 2* who is connected to *Server 2* in *Cluster 2*. *Server 1* knows that the user is not connected to its cluster since the server has a list of all users connected to the cluster and

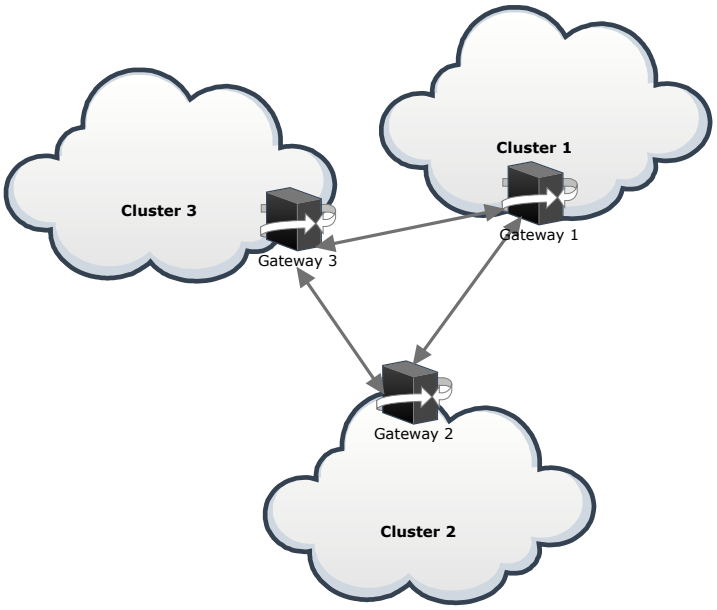

FIGURE 3.20: One Gateway Per Cloud

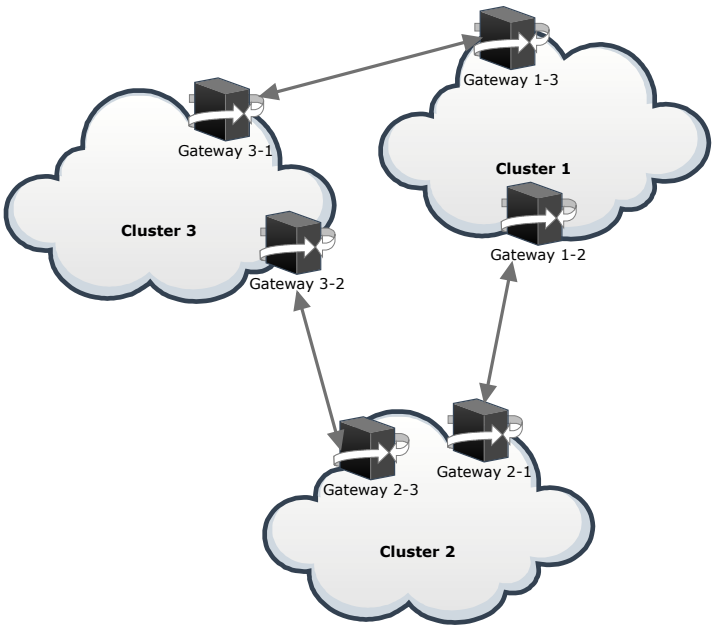

FIGURE 3.21: One Gateway Per Cloud Communication

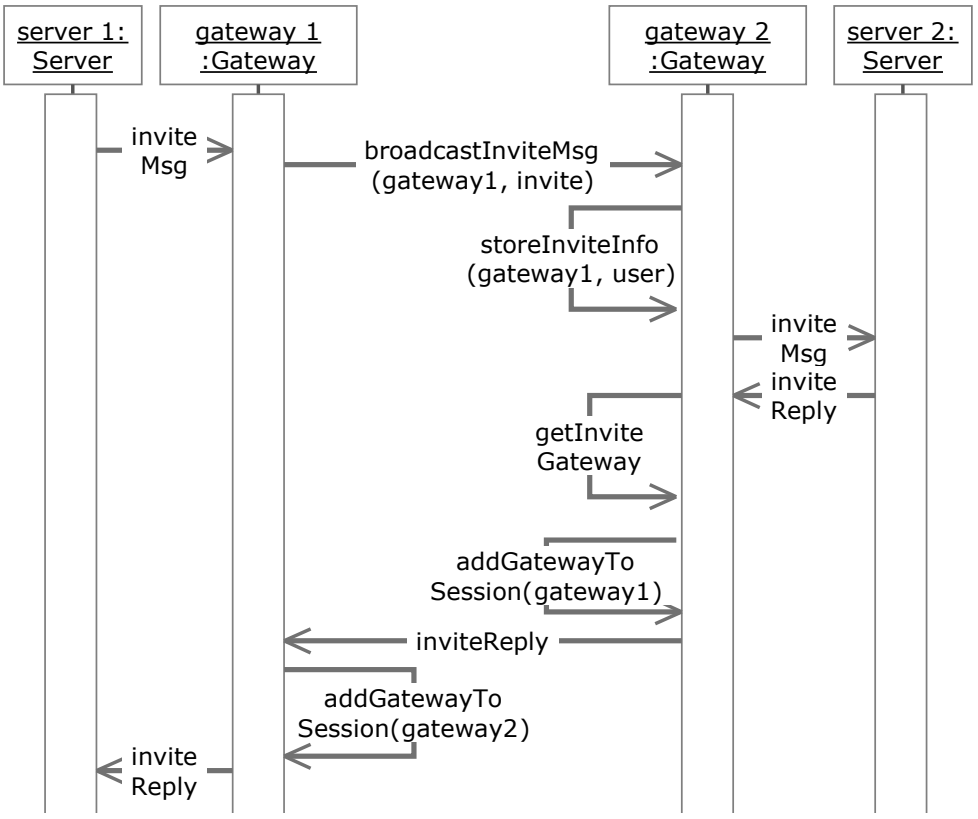

FIGURE 3.22: Session Setup Across Clouds

as such sends the invite request to the gateway. Upon reception, the gateway broadcasts the message to all known external gateways. *Gateway 2* receives the broadcast and determines that *Client 2* is connected to *Server 2* in its local cluster so it saves the invitation information to be able to route the answer back to the originating cluster and routes the message to the appropriate server. Other gateways which receive the message, determine that the required client is not connected to the cluster which they manage and drop the message. Upon reception of the reply from the client, the server determines that the inviter is not in the local cluster and sends the reply to the gateway. The gateway checks if the invite reply is an accept or decline, and if it is an accept message it stores the information that the cluster has one more client in the session the invite was performed in. It also stores the gateway the invitation originated from as being a member in the session and sends the reply to *Gateway 1*. *Gateway 1* also checks the type of the reply and performs the same actions as *Gateway 2* and finally routes the reply to the inviter's server. If multiple gateways per cluster are used servers broadcast the invite and invite reply to all gateways. In the case of invite reply, gateways which do not have stored data regarding the invite, simply drop the message.

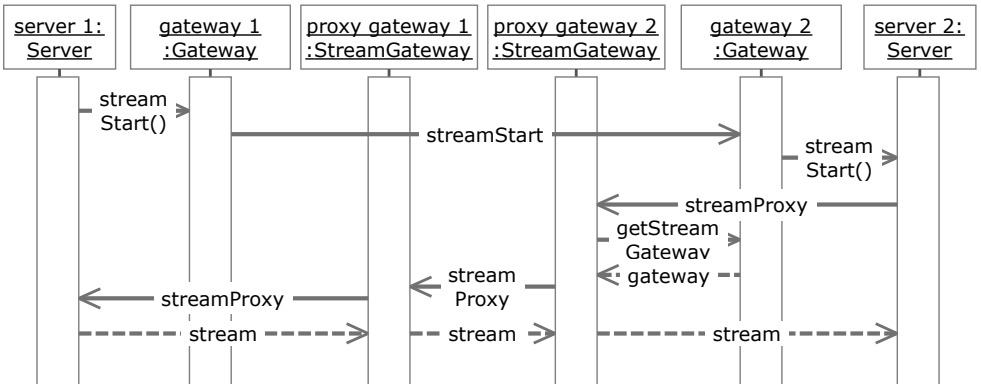

FIGURE 3.23: Stream Setup Across Clouds

The stream setup sequence is more complicated due to the introduction of the streaming proxies in the architecture. Figure 3.23 shows the sequence diagram for the stream setup. While the message that a user starts streaming goes through the gateways, the proxy request goes back through the stream gateways, which are responsible themselves for creating stream proxies. This ensures that if a user in a cloud already receives a certain stream, no proxy request will go to the originating cluster, since the stream gateway can send the stream itself. An important thing to note is that the stream gateway checks with the message gateway to see where the stream originating gateway is, and then uses the equivalent stream gateway to request the proxy. The stream proxy has to have in this case a mapping of gateways to stream gateways. Such knowledge can be useful also for the case where multiple gateways are used per cluster.

3.4.2 Cloud Admission Control

Another very important consideration for the system is the admission control used to accept and route clients to servers. A single cluster deployment can use a very simple admission control policy similar to round-robin admission. The goal of the geographic clouds deployment is to improve the user perceived performance by connecting the user to a cloud which offers better performance. Because of this requirement, the cloud deployment must use a more complex admission control scheme. A simple way to achieve such admission, is to use location based admission control. Upon reception of a connection request, the admission system determines the user's latitude and longitude by using a geolocation system and based on this information finds the closest cloud with the service. While very simple to implement, this approach does not guarantee good latency between server and

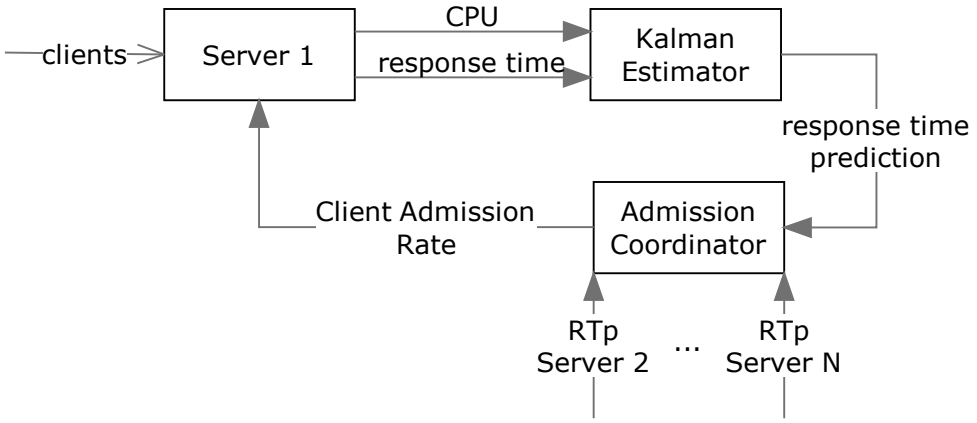

FIGURE 3.24: Server Control Loop Block Diagram

client, first of all because short geographic distance does not necessary mean short internet route and second of all because even if a short route was guaranteed, a short route could be more congested than a longer route.

A second approach to offer admission control is to allow the client to ping a small subset of close clouds and then choose the best one. This has the advantage of improving the latency between server and client at the expense of a longer start-up for the application, which must wait for the reply from the various clouds offered by the admission control system. This approach is the one used in this thesis proposal.

This approach can be seen in Figure 3.24. The Admission Controller receives the information from all the servers in the cloud, seen as RTp Server X in the figure, and uses this information to control the admission rates of the servers.

3.5 Containerization

In recent years, containers have become a very important technology as they allow for very quick deployment of applications as well as good network isolation. Containers offer almost all the benefits of virtual machines and resolve some of the issues of VMs. For this reason, the architecture described in the previous section uses containers to deploy the various servers and components. Each of the components previously described is deployed in a container, with all containers having a base container they extend which defines the OS, Java version and server version.

At the same time, two virtual networks were defined and used - the first network allows external access to the servers and only the ports which should be accessible from outside are connected to this network. The second network is internal and allows the servers to communicate with each other, this is used primarily for the JGroups communication between servers.

On top of the base container, a number of other containers were built:

1. JGroups container - container running the JGroups gossip router. It connects only to the internal network. A single container is run in a cloud but more could be added for better resiliency.

2. Media server container - includes the media server and connects to the external virtual network to allow clients to connect to it and also to to the internal network to communicate with other servers via JGroups.

3. Gateway container - includes the gateway to communicate with other clouds. Connects to the internal network as it receives messages from media servers, as well as to the external network where it exposes the port used to communicate with other gateways.

4. Admission Controller container - which routes client connections to media servers in the cloud. This container connects internally and communicates with the existing media servers to know which servers are up and which are down and also externally to receive client requests which are then routed internally.

Figure 3.25 shows the containers deployment and networks with two clouds and two media servers in each cloud.

This chapter has presented the application which is to be self-optimized through a self-organizing system and showed how a real-time collaborative application can be scaled from a single server to a geographically distributed cloud. The next chapter presents the architecture for the controller and for the self-organized autonomic system.

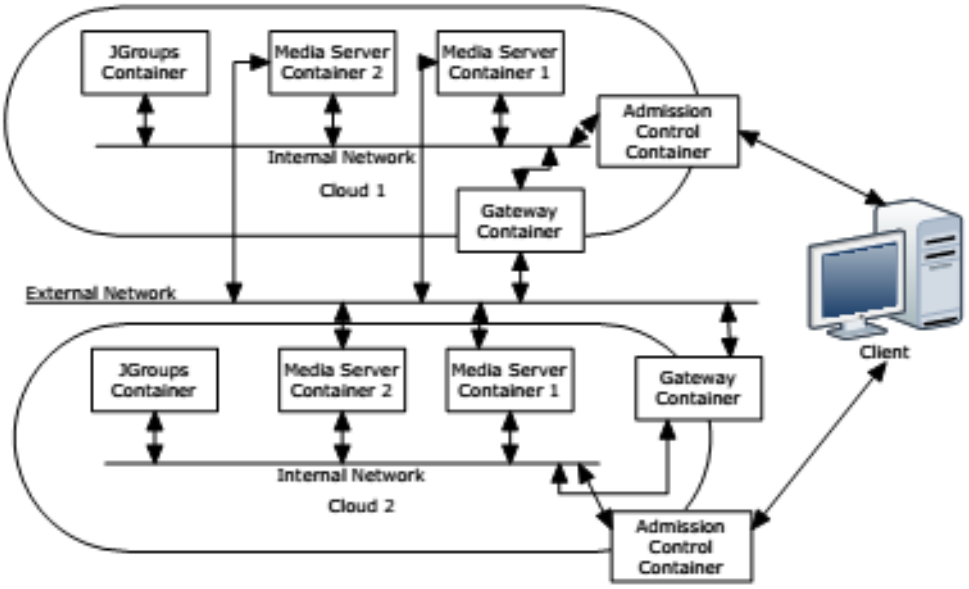

FIGURE 3.25: Stream Setup Across Clouds

Chapter 4

Self-Organizing Algorithms for Resource Control in Autonomic Systems

The next step in the development of the self-organizing autonomic system is to develop the algorithms which will be used in order to control the resources (servers) in the autonomic system. This chapter will present the self-organizing algorithms which were chosen and implemented for the control of the geographically distributed cluster of servers.

4.1 Collaborative Geographic Cluster Self-Organizing Autonomic System

The collaborative server cluster will use a self-organizing approach in order to manage the distribution of clients across the cluster as well as to scale the cluster up and down. The self-organizing system can not be based on a model which considers only the number of clients as a perturbation. Load for a collaborative server is a combination of the number of clients connected to the server, the number of sessions running on the server and most importantly the number of streams received by the server and multiplexed towards receiving clients. Based on these perturbations, each server can decide if it should receive more clients or not, independent of any decision made by another server. On top of the server

self-organization with relation to accepting new clients, the system also needs an approach to determine when the cluster's SLA will be breached and take reactive action by adding or removing servers from the cluster.

The reason to use a self-organizing systems for the adaptation is due to the intrinsic properties that self-organizing systems poses:

1. Adaptable - the ability to deal with changes in the environment which were not predicted at design time. This is important for the system developed for the thesis as we do not know the bounds of how many servers could be in a cluster at one time or the maximum peak demand for the service and as such we want a system which is able to adapt the control law dynamically.

2. Resilient - parts of the system can die or be lost but the remaining still perform their goal. In a cloud environment this is desired, as servers can come up an down at any time, and even network connectivity could be lost between servers or between data centres.

3. Emergent - the complex behaviour arises from the properties and behaviour of the simple parts. As a cloud of servers increases in size it becomes more complicated for a single controller to manage the entire system. As such a system where the control emerges from the interactions of a lot of small parts is desired as it decreases complexity and can scale much more, at the cost of overall resource utilization.

4. Anticipation - the system can anticipate problems and solve them before they impact the whole system. This is obviously desired, as we wish to detect SLA breaches before they happen, such that corrective actions can be taken and the breach can be avoided.

4.2 Server self-optimization

The first level of self-optimization for the system under control is each server's self-optimization. For the purpose of this thesis, the server self-optimization is quite simple: each server decides when to accept clients and when not to accept clients based on it's local knowledge and local control system. The control system used for each server is a fuzzy system which predicts the load on the server and

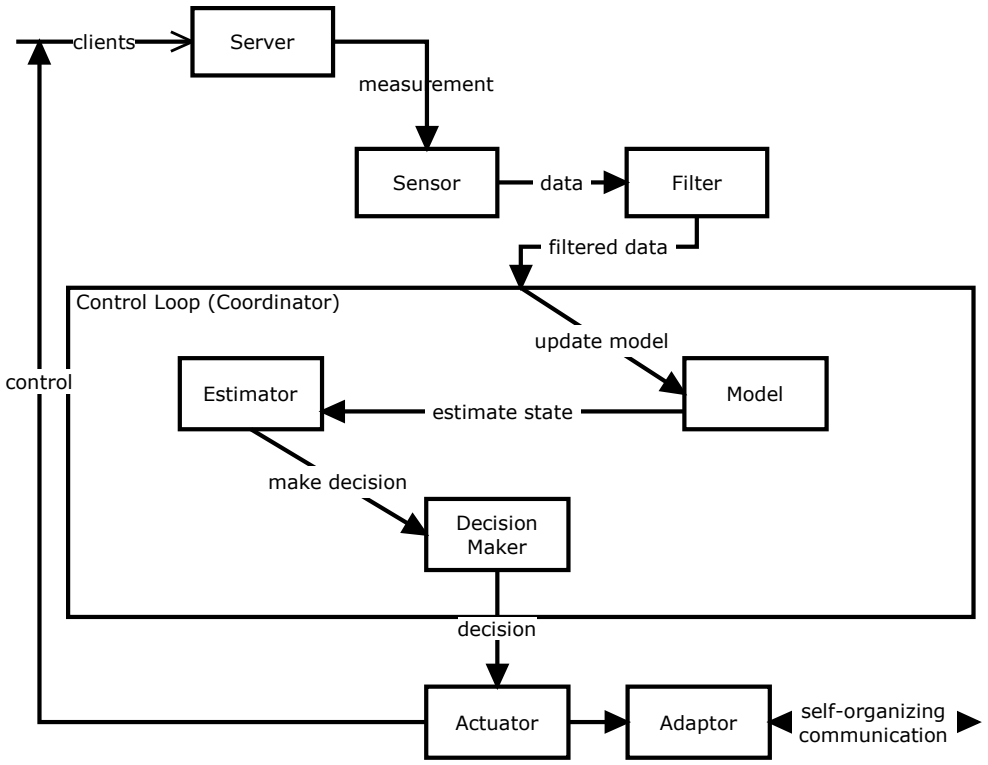

FIGURE 4.1: Self-Optimizing Control Loop Architecture

based on the predicted load either stops accepting clients or starts accepting clients again if it had stopped before. For this purpose, each server has its own control loop, similar to the one developed in **??**. The architecture can be seen in Figure 4.1. More complex control systems based on control theory can be used, however the control system for one server is not the main purpose of this thesis.

In the case of the geographically distributed cloud each server can be seen as having one perturbation signal which is the arrival of requests from clients connected to the server. Some requests, like video streaming, are more demanding on the server than others. As requests are processed, the CPU usage and response time of the server will change. More requests or more demanding requests will lead to higher CPU usage and an increase in response time. As the number of requests decreases, the CPU usage and response time will also decrease. Based on the measured sensor data, the control loop uses a fuzzy model and estimator to predict the response time for the next sampling period. This prediction is used in order to decide if the server should accept connections or not. The block diagram can be mapped to the eight component based architecture through the following steps:

1. Implement a sensor which measures the desired data from the media server. The measured data includes:
   (a) CPU usage
   (b) Number of connected clients
   (c) Number of collaborative rooms
   (d) Number of clients connected to all media servers in the cloud
   (e) Network latency
   (f) Bandwidth up/down
   (g) Number of streams incoming/outgoing from server
2. Add a filter which computes the numbers of packets in and out from the bandwidth up/down
3. Add a fuzzy model which calculates the confidence that the server is overloaded. The fuzzy model computes two fuzzy functions:
   (a) Compute confidence1 based on number of clients, packets out, packets in and number of incoming streams.
   (b) Compute confidence2 based on packets in and CPU usage
   (c) Take the maximum value of the two confidences
4. Add an estimator which compares the confidence from the model with two given thresholds. If the confidence is above the higher threshold increment a counter of how many times the system was above the higher threshold. Similarly if the confidence is bellow the threshold increment a counter of how many time the system was bellow the lower threshold.
5. Add a decision maker which uses the count of how many times the system was above/bellow the thresholds in the estimator to decide if the server should accept or reject connections. If the server is above the higher threshold for more than a given count, then the server stops accepting new clients. If the server is bellow the lower threshold for a given amount of time and is not accepting clients, then it starts accepting clients.
6. Add an actuator which communicates with the cloud load balancer. If the server should not accept connections then the actuator sends a message to the load balancer that it should stop redirecting new clients to it. Conversely,

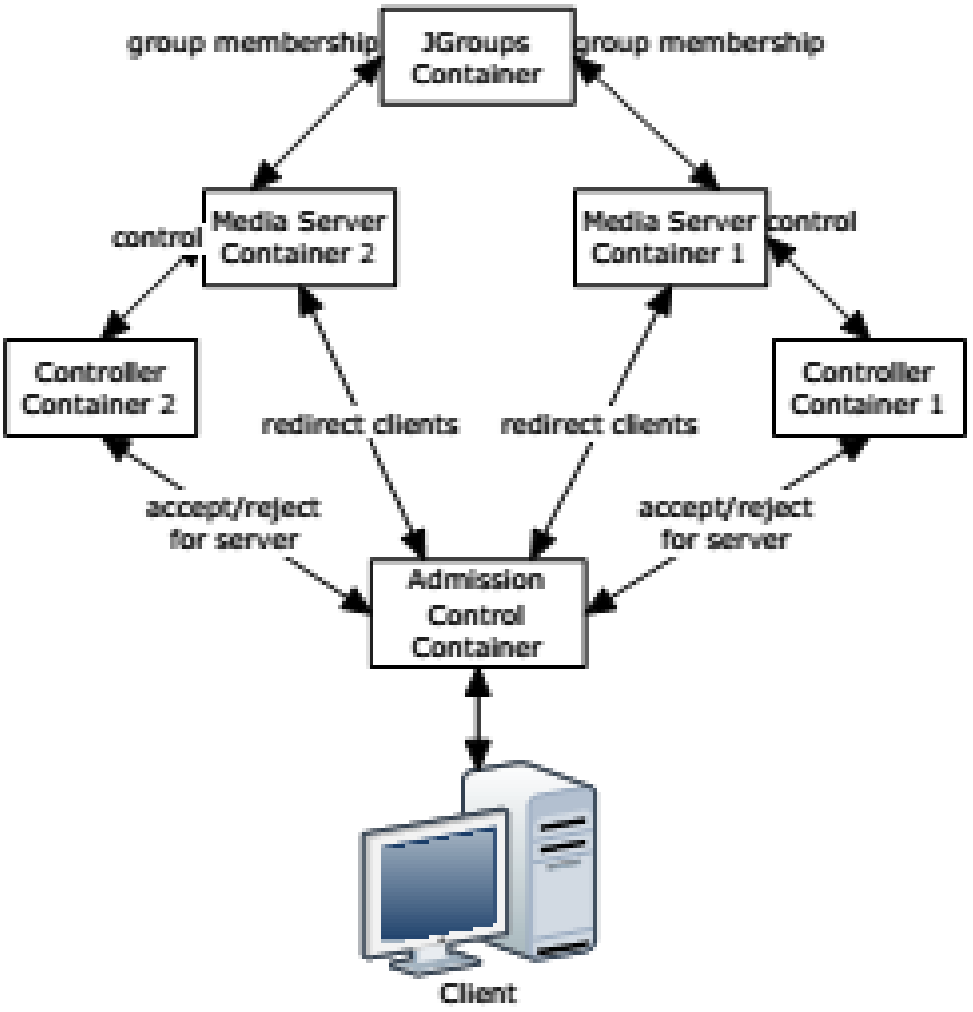

FIGURE 4.2: Control System

if the server should start accepting clients again, then it sends a message that the load balancer can start redirecting clients to it.

7. The coordinator simply coordinates the control loop and passes messages between model, controller, decision maker and actuator.

The control system for each media server is an application separated from the media server and deployed in a separate container from the media server. It can connect to the media server to retrieve sensor data but has no other interactions with the media server. The control system can also act on behalf of the media server to ask the load balancer to redirect or stop redirecting new clients to the media server. The overall control architecture looks as Figure 4.2

4.3 SLA breach detection: Ant Colony Optimization

The ant colony optimization (ACO) algorithm [22], [**?**] is best known for load balancing clouds or finding the best route in a network. The algorithm has received a lot of attention and work in academia and was presented in chapter 2. At a high level the algorithm uses a number of simple agents called ants which traverse the network and leave a trail of pheromones which other ants can then follow. Good paths or solutions are reinforced by having more ants traverse them and leaving more pheromones. Once a path or solution is no longer good, less ants travel it and another path is reinforced as more ants travel that path. In networking optimization ants behave as normal packets and travel the network from router to router. When an ant reaches a router it can look at how long the arrival router's buffer queue is for the router the ant came from and then reinforce that route based on latency, buffer sizes, etc. as shown in Figure 4.3

For the self-organizing self-optimizing control system for the collaborative media application cloud presented in this thesis, the pheromone level in the network of servers is used as a proxy for how overloaded the entire cloud of servers is and used to decide when to add or remove servers. Whenever a media server starts, the media server's control system creates a new ant and sends it through the network of servers. If the server is the first server and has no knowledge of other servers it sends no ant. Thus the total number of ants is equal to $N - 1$ where N is the number of servers. When an ant is first created, because it has no prior knowledge, it is sent to another server randomly.

As ants reach other servers they deposit pheromone at the server they arrive at, at a rate inversely proportional to the load of the server. At the same time ants wait at the server a time proportional to the load of the server. Thus overloaded servers will have less pheromone deposited when compared to an underloaded server. By having ants wait a longer time at overloaded servers, the overall amount of pheromone in the network will further decrease. With this approach, less pheromone in the network means that the cloud has more load as more servers have low pheromone levels because of being overloaded.

Assume that an ant k_1 deposits an amount of pheromone τ_{k1} when it reaches a node which has 0 load - represented by a high fuzzy confidence value, and waits at an

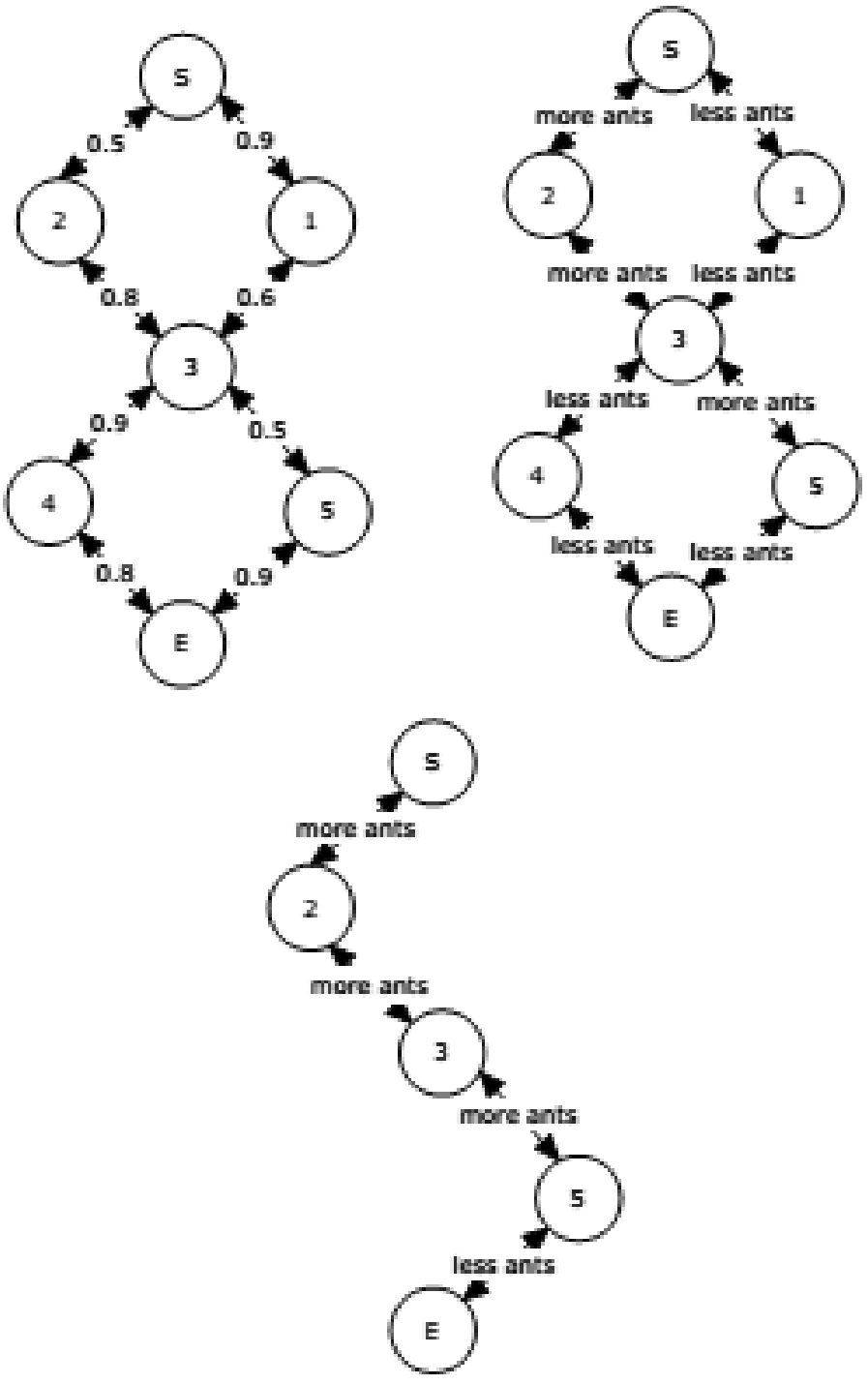

FIGURE 4.3: Ant Colony Optimization

Server	Time since last visit (s)	Pheromone Level
Server 1	15	10
Server 2	20	5
Server 3	5	8
Server 4	35	5.5
Server 5	0	10

TABLE 4.1: Ant routing knowledge prior

under-loaded node $15s$. Another ant k_2 which reaches a node where the confidence value is 50% of the higher threshold will deposit only $\tau_k * (1-p)$ where p is the fuzzy confidence as a percentage and waits at the node a time of $15s/(1-p)$ with a maximum wait time of $60s$ to avoid waiting an infinite time when p approaches 100%. At the same time, the pheromone left by the ants decays at a rate of ρ every $15s$. As such, the amount of pheromone at any node can be seen as:

$$p_n^t = p_n^{t-1} + \sum_{i=1}^{K}(\tau * (1-p)) - \rho \tag{4.1}$$

where p_n^t is the amount of pheromone at node n at time t where t can be considered discrete in $15s$ increments and K is the amount of ants arriving at the node in the time frame between $t-1$ and t.

At the same time, ants store information about which servers they have visited and time passed since the last visit. When an ant decides which server to go to, it uses a random function which is proportional to the time since it has not visited a server combined with the pheromone level of the destination server. Thus the ant will give preference to the servers it has not visited in a long time, and especially the servers it has never visited. Because the server structure is not stable and servers can join and leave at any time, ants decide the next server to visit based on the servers known by the current server the ant is at. Assume a cloud with 5 servers and an ant which has the following information in it's visit history table and which reaches Server 5.

Based on the table, the ant computes the probability of visiting each server as:

$$P_s = (t_s / \sum_{i=1}^{N} t_i + p_s / \sum_{i=1}^{N} p_i)/2 \tag{4.2}$$

Server	Probability (%)
Server 1	35.10
Server 2	26.48
Server 4	38.42
Server 5	0

TABLE 4.2: Ant routing probability

Server	Time since last visit (s)	Pheromone Level
Server 1	0	8
Server 2	25	5
Server 3	10	8
Server 4	40	5.5
Server 5	5	10

TABLE 4.3: Ant routing knowledge posterior

where P_s is the probability of visiting server s, t_s is the time since it has visited server s last time and p_s is the pheromone at server s. These probabilities are computed only on the servers known as being up by the server the ant is at. The server the ant is currently at has to be excluded from the calculations, and it's probability will be 0. Let us assume also that Server 5, which is the server the ant is at currently does not yet know about server 3. As such, the visit probability table looks as follows:

As such, the ant will roll a random value between 0 and 1, and choose which server to go to. A value between 0 and 0.3842 means Server 4, between 0.3842 and 0.7352 means Server 1 and between 0.7352 and 1 means Server 2. Assuming the value the ant rolls is 0.7, and that the ant waits at Server 5 for 5s, the routing table after the ant moves to the next server will be:

In order to avoid having all ants visit a new node at the same time, whenever an ant discovers a new server it initializes the time since it visited the server with a random value. Assume that after the ant reaches Server 1, it discovers a new server which was unknown before - Server 6. The time since last visiting Server 6 will be initialized with a random value between 0 and the maximum time since last visiting a server which is known to be alive as in Equation 4.3. Furthermore the known pheromone level of the new server will be 0. If Server 1 only knows Server 2, 5 and 6 then the random value will be between 0 and 25s.

$$t_{new} = random(0, max(t_{known})) \tag{4.3}$$

The ACO algorithm is used in order to decide when the system is about to breach its SLAs. Once the ACO algorithm detects that the system's SLA is about to be breached the ant relocation algorithm starts in order to find how many servers should be started or removed. As such, each ant is also responsible for measuring the pheromone level at each node it visits. Ants store a history of the last N nodes that they visited and the pheronomone level at each of these nodes. Every time an ant moves to a new server, it removes the oldest pheronomone value from this list and adds the pheronomone at the current node. It then computes an aggregate metric of the pheronomone at the last N nodes visited, which is a simple average:

$$p_{ant} = \sum_{i=1}^{N} p_N/N \tag{4.4}$$

where p_N is the pheromone at server N. Once the pheromone level of an ant goes under a predefined value, the ant morphs into a house hunting ant which searches for the new count of servers which should be added to the cloud. Similarly, if the pheromone level of an ant becomes too high, the ant morphs and starts searching for the number of servers which should be removed from the cloud. The house hunting algorithm used to optimize the number of servers in the cloud does not start until enough of the ants have morphed into house hunting ants. Until enough ants have morphed for the house hunting algorithm to start, the morphed ants continue behaving as ACO ants. Once more than half of the ants morph all ants go to one node and the house hunting algorithm starts.

When an ant reaches a new control node it is processed by the controller as shown in Figure 4.1

Algorithm 4.1 Ant Colony Optimization Pseudocode

```
Calculate pheromone at current node
Update ant's server tables
if Ant should morph then
    Morph ant
else
    Calculate next server for ant
    Send ant to next server
end if
```

For example, for a network of five servers figure 4.4 shows the behaviour of the ant algorithm.

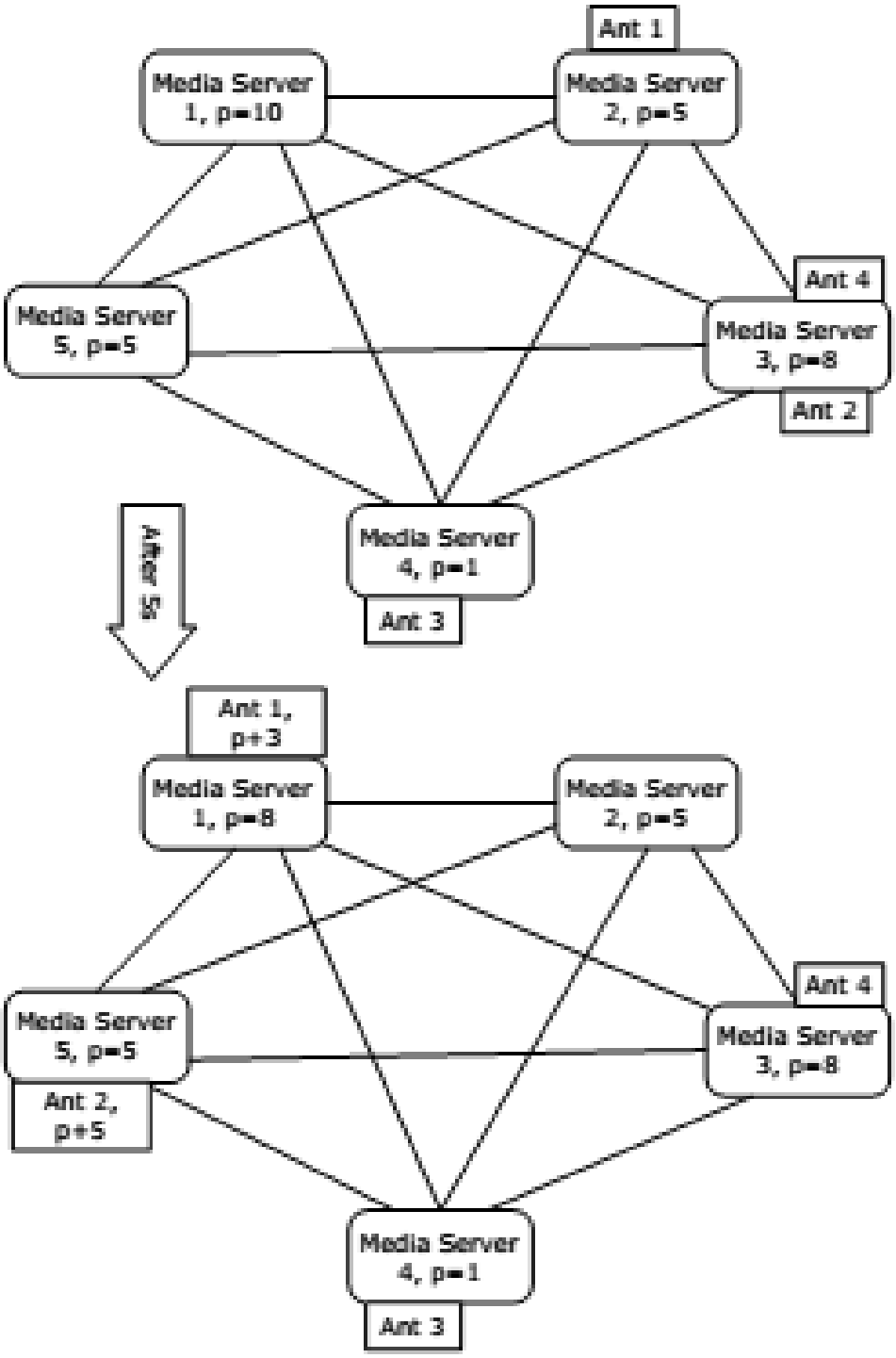

FIGURE 4.4: Ant Colony Optimization Network

4.3.1 Cluster optimization: Ant house hunting

The ant house hunting algorithm can be used in order to optimize the number of servers in the cloud by having ants search for the optimal number of servers which should be up in the cloud. This algorithm works by having each ant search for a new nest, where a nest is represented by a new optimum number of servers in the cloud. The algorithm assumes that there is a home nest where the ants can meet after searching for a new nest. Once more than half of the ants have agreed on the same optimum value, the servers are added or removed as desired and all ants are morphed back into food foraging ants for the ACO algorithm. [**?**]

When an ant morphs into a house hunting ant it initializes itself with a range of possible solutions for the new size of the cloud. These possible solutions are computed as permutations of the current size of the cloud as known by the server the ant is at. If the ant was morphed because of lack of pheromone, then the ant generates a random value which would represent the number of servers to be added. Each random value is proportional to the size of the cloud and the pheromone level across the last N servers known by the ant. For example, if the cloud contains only 2 servers, then the ant is initialized with a random value between 1 and 2. However, if the cloud contains 100 servers, then the ant would be initialized with a random values, which is defined as:

$$ServerCount_{add} = 1 + rand * N/(p_{ant}) \tag{4.5}$$

such that the lower the pheronomone value, the higher the probability of more servers being added. The random value has the effect of generating multiple possible solutions across different ants.

If however the ant was morphed because of too much pheromone, then it does a similar initialization, but in this case it randomly chooses X servers and hides them from its knowledge of the cloud. The servers are still in the ant's history but it does not consider them for optimization calculations. Each ant will create a different such permutation of servers which are hidden.

The ant house hunting algorithm is performed in rounds and ants can be in one of four states:

1. Search - This is the initial state of ants which search for a new nest

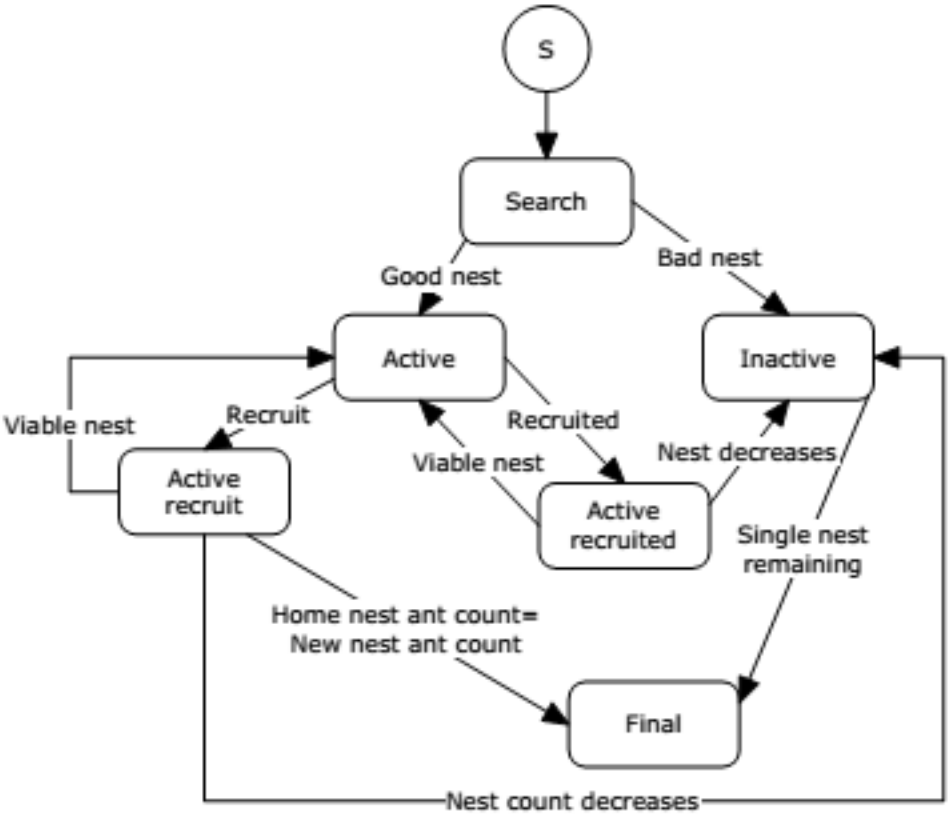

FIGURE 4.5: Ant House Hunting State Diagram

2. Active - The ant is committed to a good nest and tries to recruit other ants to it

3. Passive - The ant is committed to a bad nest and is waiting to be recruited

4. Final - A single nest remains so all ants go to it and that is the solution

The state diagram for the ants can be described as in Figure 4.5

In the first round all house hunting ants search for a new nest by searching their solution space. This is achieved by having each ant choose one solution and simulating its effect on the pheromone level. If the ant is a server adder then the ant simulates that the $ServerCount_{add}$ servers are up and with no load. It picks K servers from the list of the last N servers it has visited and hides them, and replaces them with servers with no load. It then uses this information in order to simulate the pheromone level. If the ant is a subtractor, it hides K servers where K is defined as part of its random solution and simulates the effect this would have on its pheromone level. If the simulated result's performance is under

a given threshold, then the ant goes into the passive state, otherwise it goes into the active state.

In the second round, all ants return home and active ants try to recruit other ants at the home nest. Passive ants can not be recruited until the final round when all ants go to the single remaining nest. Active ants recruit randomly from the other active ants at the home nest by choosing another ant to recruit and bring it to it's committed nest. In order to ensure no conflicts between recruiting ants, the ants recruit iteratively and an ant which was already recruited does not recruit someone else. Once an ants recruits another ant, then the two ants go together to the nest of the recruiting ant.

When reaching a new nest, active ants count the number of ants at the nest and check if the nest they reached is the same as the previous nest they went to. Based on the number of ants and the nest there are three cases:

1. If the nest is the same nest that the ant went to before and the number of ants has increased or remained the same then the nest is still a possible solution so the ant updates the count and waits an extra round at the nest. After waiting a round, the ant checks if the number of ants at the home nest is the same as that at it's current nest. If they are the same then the ant goes to the final state, new servers are added or removed and all ants morph back to ACO ants.

2. If the nest is the same nest that the ant went to before and the number of ants has decreased then the ant becomes passive because the nest is in the process of being dropped out, and returns home in the same round that active ants wait at the new nest.

3. If the nest is different then the nest the ant was committed to then it checks to see if the number of ants at it's new nest has decreased or not. If the number has decreased, then this nest is dropping out as the ants already committed to it have gone to the home nest and the ant becomes passive. Otherwise, the ant commits to it's new nest and goes home to recruit other ants.

The above algorithm can be described as follows in pseudo-code in Figure 4.2. R_x represents round x.

Algorithm 4.2 Ant House Hunting Pseudocode

R_1: Go to new nest
Compute suitability of new nest
if Suitability ¡ threshold **then** Switch to passive
end if
R_2: Go to home nest
if Ant is active && Ant is not recruited **then**
    Recruit another ant
    Go to new nest
else if Ant is recruited **then**
    Go to new nest
end if
R_3: Count number of ants at new nest = $count_{new}$
if Nest is same and $count_{new} >= count_{old}$ **then**
    Wait round
else if Nest is same and $count_{new} < count_{old}$ **then**
    Switch to passive
    Go to home nest
else if Nest is different **then**
    Wait round
end if
R_4: Count number of ants at new nest $count_{new}$
if $count_{new} == count_{home}$ **then**
    Switch to final state
else if $count_{new} < count_{old}$ **then**
    Switch to passive
    Go to R_2
end if
Return final state
Switch ants to ACO ants

4.4 Self-organizing algorithm conclusions

The proposed approach for the self-optimizing control of the geographically distributed multimedia cloud makes use of two self-organizing algorithms. The first algorithm is used in order to detect breaches of SLA before they happen such that proactive measures can be taken. The second self-organizing algorithm finds the optimal count of servers which should be running in the cloud.

The thesis uses the ACO algorithm to detect possible breaches of SLA by having ants continously move through the network and measure the load on the network of servers. This is a novel approach as previous reasearch has only focused on using the ACO to optimize various problems and not for detection of possible

problems. The proposed approach ensures that the detection of SLA breaches can be done ahead of the actual breach while at the same time being fully scalabe as the self-organizing algorithm scales with the number of servers.

The second algorithm which optimizes the number of servers in the cloud makes use of a different self-organizing approach also inspired by the behaviour of ants. This algorithm is also novel as previous research into the house hunting algorithm was only used as a way to model the behaviour of ants and not for problem optimization.

The following chapter will introduce the test bed used to test the algorithms and also present results for the implementation of the self-optimizing self-organizing geographically distributed media cloud.

Bibliography

[1] Akamai, "Akamai CDN," http://www.akamai.com, Akamai, [Accessed: May 2014]. [Online]. Available: http://www.akamai.com

[2] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," IBM Corp., October 2001, [Accessed: May 2009]. [Online]. Available: http://researchweb.watson.ibm.com/autonomic/

[3] K. Rohloff, R. Schantz, and Y. Gabay, "High-level dynamic resource management for distributed, real-time embedded systems," in *SCSC: Proceedings of the 2007 summer computer simulation conference*. San Diego, CA, USA: Society for Computer Simulation International, 2007, pp. 749–756.

[4] M. Litoiu, "A performance analysis method for autonomic computing systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 2, no. 1, pp. 1–27, 2007.

[5] 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*, 2007, pp. 1–10.

[6] D. Devescovi, E. Di Nitto, and R. Mirandola, "An infrastructure for autonomic system development: the selflet approach," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 449–452.

[7] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "A multi-agent systems approach to autonomic computing," in *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 464–471.

[8] W. R. Ashby, *Design for a brain*. Wiley New York, 1954.

[9] 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.

[10] B. Solomon, D. Ionescu, C. Gadea, S. Veres, and M. Litoiu, "Self-optimizing autonomic control of geographically distributed collaboration applications," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC '13. New York, NY, USA: ACM, 2013, pp. 28:1–28:8. [Online]. Available: http://doi.acm.org/10.1145/2494621.2494650

[11] 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.

[12] B. Solomon, D. Ionescu, M. Litoiu, and G. Iszlai, "Designing autonomic management systems for cloud computing," in *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, may 2010, pp. 631 –636.

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

[14] IBM, "An architectural blueprint for autonomic computing," White paper, IBM, June 2006, [Accessed: May 2009]. [Online]. Available: http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

[15] M. M. Fuad and M. J. Oudshoorn, "System architecture of an autonomic element," *Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, pp. 89–93, March 2007.

[16] IBM, "IBM - IBM Tivoli Intelligent Orchestrator - Tivoli Intelligent Orchestrator - Software," IBM, 2013, [Accessed: August 2013]. [Online]. Available: http://www-947.ibm.com/support/entry/portal/Overview/Software/Tivoli/Tivoli_Intelligent_Orchestrator

[17] "Sun N1 Service Provisioning System," SUN, 2013, [Accessed: August 2013]. [Online]. Available: http://docs.oracle.com/cd/E19118-01/n1.sprovsys52/

[18] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–268. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.19

[19] E. Gat, "On three-layer architectures," Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems, 1998, [Accessed: November 2011]. [Online]. Available: http://www.flownet.com/gat/papers/tla.pdf

[20] M. Litoiu, M. Woodside, and T. Zheng, "Hierarchical model-based autonomic control of software systems," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[21] A. A. Rhissa and A. Hassnaoui, "Towards a global autonomic management and integration of heterogeneous networks and multimedia services," *Network Control and Engineering for QoS, Security and Mobility, IV*, pp. 199–207, May 2007.

[22] C.-H. Chu, J. Gu, and Q. Gu, "A heuristic ant algorithm for solving qos multicast routing problem," in *CEC '02: Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress.* Washington, DC, USA: IEEE Computer Society, 2002, pp. 1630–1635.

[23] R. Albert and A.-L. Barabasi, *Statistical mechanics of complex networks.* Reviews of Modern Physics, 2002.

[24] A. Montresor, H. Meling, and z. Babaolu, "Messor: Load-balancing through a swarm of autonomous agents," in *Agents and Peer-to-Peer Computing*, ser. Lecture Notes in Computer Science, G. Moro and M. Koubarakis, Eds. Springer Berlin / Heidelberg, 2003, vol. 2530, pp. 125–137.

[25] J. Xu, M. Zhao, and J. A. Fortes, "Cooperative autonomic management in dynamic distributed systems," in *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 756–770. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05118-0_52

[26] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 592–603. [Online]. Available: http://dl.acm.org/citation.cfm?id=1546683.1547490

[27] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proceedings of the 2009 conference on Hot topics in cloud computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855533.1855545

[28] R. Zhang and A. J. Bivens, "Comparing the use of bayesian networks and neural networks in response time modeling for service-oriented systems," in *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*. New York, NY, USA: ACM, 2007, pp. 67–74.

[29] J. Joyce, "Bayes' theorem," Metaphysics Research Lab, CSLI, Stanford University, September 2003, [Accessed: May 2009]. [Online]. Available: http://plato.stanford.edu/entries/bayes-theorem/#1

[30] M. B. Sheikh, U. F. Minhas, O. Z. Khan, A. Aboulnaga, P. Poupart, and D. J. Taylor, "A bayesian approach to online performance modeling for database appliances using gaussian models," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998603

[31] D. Ardagna, M. Trubian, and L. Zhang, "SLA based resource allocation policies in autonomic environments," *Journal of Parallel and Distributed Computing*, vol. 67, no. 3, pp. 259–270, 2007.

[32] R. Calinescu and M. Kwiatkowska, "Using quantitative analysis to implement autonomic it systems," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 100–110. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070512

[33] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow," in *Seventh International Conference on Autonomic and Autonomous Systems, ICAS 2011*. IEEE, May 2011, pp. 67–74, moVe INT LIP6.

[34] P. Martin, M. Zhang, W. Powley, H. Boughton, P. Bird, and R. Horman, "The use of economic models to capture importance policy for autonomic database management systems," in *Proceedings of the 1st ACM/IEEE workshop on Autonomic computing in economics*, ser. ACE '11. New York, NY, USA: ACM, 2011, pp. 3–10. [Online]. Available: http://doi.acm.org/10.1145/1998561.1998564

[35] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin, "What does control theory bring to systems research?" *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 62–69, January 2009. [Online]. Available: http://doi.acm.org/10.1145/1496909.1496922

[36] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva, "Decision making in autonomic computing systems: comparison of approaches and techniques," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 201–204. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998629

[37] X. Liu, X. Zhu, S. Singhal, and M. Arlitt, "Adaptive entitlement control of resource containers on shared servers," in *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, may 2005, pp. 163 – 176.

[38] C. Lu, Y. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, "Feedback control architecture and design methodology for service delay guarantees in web servers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pp. 1014–1027, September 2006. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2006.123

[39] W. Pan, D. Mu, H. Wu, and L. Yao, "Feedback control-based qos guarantees in web application servers," in *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, sept. 2008, pp. 328 –334.

[40] G. Anders, C. Hinrichs, F. Siefert, P. Behrmann, W. Reif, and M. Sonnenschein, "On the influence of inter-agent variation on multi-agent algorithms solving a dynamic task allocation problem under uncertainty," in *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on*, 2012, pp. 29–38.

[41] N. M. Calcavecchia, B. A. Caprarescu, E. D. Nitto, D. J. Dubois, and D. Petcu, "Depas: a decentralized probabilistic algorithm for auto-scaling," *Computing*, vol. 94, no. 8-10, pp. 701–730, 2012.

[42] B. A. Caprarescu, N. M. Calcavecchia, E. D. Nitto, and D. J. Dubois, "Sos cloud: Self-organizing services in the cloud," in *BIONETICS*, 2010, pp. 48–55.

[43] F. Dotsch, J. Denzinger, H. Kasinger, and B. Bauer, "Decentralized real-time control of water distribution networks using self-organizing multi-agent systems," in *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, 2010, pp. 223–232.

[44] E. Feller, L. Rilling, and C. Morin, "Snooze: A scalable and autonomic virtual machine management framework for private clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 2012, pp. 482–489.

[45] B. A. Caprarescu, "Robustness and scalability: a dual challenge for autonomic architectures," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ser. ECSA '10. New York, NY, USA: ACM, 2010, pp. 22–26. [Online]. Available: http://doi.acm.org/10.1145/1842752.1842759

[46] E. Feller, C. Rohr, D. Margery, and C. Morin, "Energy management in iaas clouds: A holistic approach," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 204–212.

[47] Apache, "Google Wave Protocol," Apache, 2013, [Accessed: August 2013]. [Online]. Available: http://www.waveprotocol.org/wave-in-a-box

[48] C. A. Gutwin, M. Lippold, and T. C. N. Graham, "Real-time groupware in the browser: testing the performance of web-based networking," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, ser. CSCW '11. New York, NY, USA: ACM, 2011, pp. 167–176. [Online]. Available: http://doi.acm.org/10.1145/1958824.1958850

[49] JGroups, "JGroups - A Toolkit for Reliable Multicast Communication," JGroups, 2013, [Accessed: August 2013]. [Online]. Available: http://www.jgroups.org/