uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Bogdan Vlad Solomon

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (Electrical Engineering)

GRADE / DEGREE

Department of Electrical Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A Real-Time Pattern Based Architecture for Autonomic Systems

TITRE DE LA THÈSE / TITLE OF THESIS

W. Gueaib

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

A. El Saddik

D. Petriu

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

UNIVERSITY OF OTTAWA

A Real-Time Pattern Based Architecture for Autonomic Systems

Bogdan Solomon

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

Your file Votre référence
ISBN: 978-0-494-58219-0
Our file Notre référence
ISBN: 978-0-494-58219-0

"Everything should be made as simple as possible, but not simpler."

Albert Einstein

UNIVERSITY OF OTTAWA

Abstract

Faculty of Graduate and Postdoctoral Studies
The Ottawa-Carleton Institute for Electrical and Computer Engineering

Master's Degree

by Bogdan Solomon

Autonomic Systems are computer systems that can self-manage by ensuring that the system is optimal, correctly configured, protected and able to recover from errors with minimal to no 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. The methods used to undertake research in the area of autonomic systems are also varied, ranging from complex machine learning algorithms to simple rule-based control systems and from complex control models of the underlying system to simple policy based control systems. At present, autonomic system structures are missing a key component - a robust, reusable, and standard architecture based on patterns. This would simplify the design and implementation of autonomic computing systems. Autonomic systems are often designed and developed following a monolithic pattern which leads to a replication of development efforts when parts of the system are later used in a different approach. In this thesis, an architecture rooted in real-time control systems is introduced and its corresponding detail design is described. The development of the architecture is based on Model-Driven Development principles. Furthermore, the thesis presents and develops various models for the control of autonomic systems. The models used allow for an easy interchange of components via composition/decomposition mechanisms. It is thus demonstrated how the framework developed based on the proposed architecture can be easily reconfigured for the control and supervision of different types of autonomic computing strategies such as self-management for Web Services or self-optimization for Server Virtualization. Experiments on and with the above new autonomic computing methodology for various computing paradigms 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 work as well as for the oppurtunity of doing this Master's. 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.

I would also like to thank IBM Canada for supporting the work done and especially Dr. Marin Litoiu from the IBM Center of Advanced Studies (CAS) in Toronto, presently with York University, for the help and advice I received while working on the thesis project from the IBM Toronto CAS laboratory.

Special thanks also go to all the people in the NCCTLab and the MCRLab 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

Abbreviations

ACM	**A**ssociation for **C**omputing **M**achinery
ARM	**A**pplication **R**esponse **M**easurement
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
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
JEE	**J**ava **E**nterprise **E**dition
JMX	**J**ava **M**anagement e**X**tension
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 Standards
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 the IT departments have become more pervasive and more important to every company, as well as to our daily lives, they have also become more complex. In some cases so complex that it is nearly impossible for humans to continue to manage the IT infrastructure. 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 and even the loss of customers to competitors for a company. Even worse, an IT failure can result in a market crash where millions or billions of dollars can vanish. The crash of 1987 is an example of such a problem when a glitch in the functioning of the application which traded stocks automatically took the global economy in a state of crisis. [1]

The increase of the dependence on the web, be it searching for a map, looking for locations online, shopping or gaming online adds to the increasing need for self-managing systems. The explosion of social networks, web related services and mashups has also led to the emergence of cloud computing. Where before each small enterprise would run its own small IT department, now large data centers provide IT services to multiple consumers and enterprises. For the IT providers the ability to maintain the systems and prevent service outages is paramount since long period of failures can open them to liabilities from their customers.

On top of the complexity of managing the IT infrastructure, two new factors have appeared in recent years that compound the problem. On one hand, most businesses are attempting to become greener and reduce the amount of energy used. As such, server farms can no longer be set up to face a worse case scenario demand, and during non peak times operate at five to fifteen percent server utilization. [2], [3] The solution is to intelligently move loads such that only the required CPU power is used for a certain demand. Another factor that increases the problem is the recent push that server virtualization has seen in data centers. While on one hand, server virtualization improves the usage of servers in data centers, it also increases the complexity of managing the servers in the data centers, as suddenly one single hardware server can be running tens of virtual machines, each with its own load and processing requirements.

Because of all the above factors, in 2001 IBM launched the Autonomic Computing Manifesto [4]. The goal of autonomic computing is to build systems that are capable of self-managing themselves by self-configuring, self-healing, self-optimizing and self-protecting themselves, together known as self-CHOP. Such a system must be able to analyze itself at runtime, determine its state, determine a desired state and then if necessary attempt to reach the desired state from the current state. Normally the desired state is a state that maintains the system's Service Level Agreement (SLA). For a self-configuring system for example, this could include finding missing libraries and installing them with no human intervention. A self-healing system would be able to determine errors in execution and recover to a known safe state. A self-optimizing system example would be a cluster of servers that dynamically adds and removes servers at runtime in order to maintain a certain utilization and client response time. Finally, a self-protecting system example would be a server that detects a denial of service (DoS) attack and prevents it by refusing requests from certain Internet Protocol (IP) addresses.

The above goals of autonomic computing were mapped in the Manifesto [4] 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

With the launch of the Manifesto a varied number of research projects were started with the goal of building autonomic systems, ranging from ways of monitoring and controlling the underlying system to rules of how resources should be distributed among competing requirements. Among the control rules, the research was also varied spanning from control loop theory approaches to machine learning approaches. However, one problem that existed was that each research project and each product lived in its own hermetic world, and in many cases this lead to code being written to do the same function by multiple people. For example, if one project required the monitoring of a cluster of application servers in order to optimize the hardware usage and another required the monitoring of a cluster in order to optimize the thread pools in the servers, each project would write its own monitoring code. Because of the above there is a need for a standard architecture for autonomic computing. An architecture which would allow developers and researchers to concentrate on the problem that they are trying to solve, and not on communication with and control of external entities. Such an architecture has to be modular and contain standardized APIs in order for modules developed by various parties to be tied together at runtime in order to perform the required autonomic goal. At the same time, the system built on top of the architecture must be easily manageable through the use of a graphical user interface, which would allow the composition of parts in order to create an autonomic management system.

The solution proposed by this thesis is an architecture rooted in real-time system theory and based on software engineering patterns and practices. On top of the

proposed architecture a framework is built that supports the real time composition of modules in order to create autonomic systems. The architecture and subsequent framework are developed using the Model-Driven Development (MDD) approach. At the same time, the framework is built on top of open standards, thus meeting the requirements set by the Manifesto. [4] Furthermore, the solution allows for the composition and recomposition of autonomic modules at run time, thus creating a control system that is itself capable of reconfiguring the internal structure and execution of its logic. The framework is applied to the problem of self-optimization of clusters of servers in both virtualized and non-virtualized environments.

1.2 Organization of the Thesis and Contributions

The thesis is organized in order to present the solution moving from a high level architecture to the low level details of the architecture, going into the various real time mechanisms set up to ensure integrity of data and reliability of the system. The framework that implements the architecture is also presented, together with a solution for server optimization created on top of the framework. Results from running the solution are also shown. As such the rest of the thesis is organized as follows.

Chapter 2 presents the need for an architecture for autonomic systems and also presents various related work in both autonomic architectures and self-optimization problems. The requirements for the architecture and framework are also presented.

Chapter 3 introduces the theory behind the model built for the autonomic system, and also presents the model identification steps taken in order to initialize the model.

Chapter 4 describes the high level design of the proposed architecture. It maps the requirements for the system to the architecture's main components.

Chapter 5 goes into the low level architecture and describes the various real time mechanisms used to provide reliability to the architecture.

Chapter 6 introduces the problem of self-optimization that is solved in order to show how the architecture and framework can be used. It also describes some results obtained by running the systems presented on a test bed.

Finally chapter 7 presents conclusions of the benefits provided by the proposed solution. Future developments of the architecture are also described.

This thesis contains a number of contributions to the field of autonomic computing, as presented in [5], [6], [7], [8], [9], [10], [11], [12]:

1. A standard architecture for autonomic systems is presented which can decrease the time of development and deployment of new autonomic systems. By standardizing the architecture, using object composition to build the system and defining interfaces for the architecture's components, a flexible system can be developed which allows researchers to concentrate and improve a smaller subset of the autonomic control loop. A standard architecture can also ensure easier communication with third party components, which is one of the requirements for an autonomic system as defined by IBM.

2. Issues of reliability and concurrency in such a distributed system are discussed and solutions based on real time patterns are presented. Since the developed autonomic system architecture is the first distributed architecture for an autonomic system - all previous autonomic systems being monolithic structures, it must deal with reliability and concurrency issues, and the best solution can be found by applying real time patters to the developed architecture, since real time systems must always deal with these issues.

3. A new model for an application server's response time is created, which is based on examining the internal mechanics of the sever's internal scheduling and queuing mechanisms. Based on the above analysis of the processes executing inside an application, a system of equations is developed in order to describe the relation between the input variables of the system, the state of the system and the controlled variables of the system. On top of the developed equations an Extended Kalman Filter is added for control of the system and model identification is performed in order to determine the initial values of the model's parameters.

4. A framework based on open standards and the above architecture is also developed. The thesis presents a framework created on top of the Web Service

Distributed Management (WSDM) standard. This framework is the first framework developed for autonomic systems which is built entirely on open standards. Through the use of WSDM, the requirement for an autonomic system to be based on open standards - as described in the Autonomic Computing Manifesto [4] such that systems developed by different vendors can interoperate is achieved.

5. A solution for allowing the modification of the control system at run time is also presented. Due to the way the architecture splits the autonomic system into components, and the capabilities added by the framework and the real-time software patterns used, the control loop itself can be modified at run time with minimal impact to the execution of the autonomic system. This capability of real time adaptation in the control loop allows the autonomic system to better perform its management function as it can better respond to changes in the environment and the managed resource by modifying its own internal structure.

6. The autonomic computing architecture is applied to the control structure of two autonomic computing environments - the self-optimization of a cluster of WebSphere Application Servers and the self-optimization of a cluster of Virtual Machines with WebSphere Application Servers installed on top. Through the use of the architecture and framework, it is demonstrated how autonomic system components can be developed on top of the system described in the thesis.

Chapter 2

Autonomic Computing Systems

This chapter presents the need for a standard architecture for autonomic systems from both an academic and an industrial point of view. It then discusses related work in the field of autonomic computing and shows how the present research makes use and at the same time expands previous research in the domain. Finally the requirements for the system are presented.

2.1 The Need for an Architecture

Since the release of the Autonomic Computing Manifesto [4], autonomic system research has received a significant amount of attention from the research community. Most of the research is based on the idea of a MAPE loop (Monitor, Analyze, Plan, Execute) as described by IBM in the Autonomic Computing Blueprint [13]. Figure 2.1 shows the loop as described by IBM. The loop's starting point is the constant monitoring of the underlying resource, be it a single system like a server, a homogeneous system like a cluster of servers, a heterogeneous system like a data center including servers, switches, routers, etc., or a full business system which includes all the knowledge and systems which make up the business' infrastructure. The monitor function of the autonomic system is responsible to gather data from the underlying resource, filter, process and correlate data from the parts composing the resource and if necessary trigger a symptom which is sent to the analyze function. Once a symptom is triggered the analyze function attempts to determine if the current state or the predicted future state of the system will be outside required parameter values. If such a breach of requirements is detected,

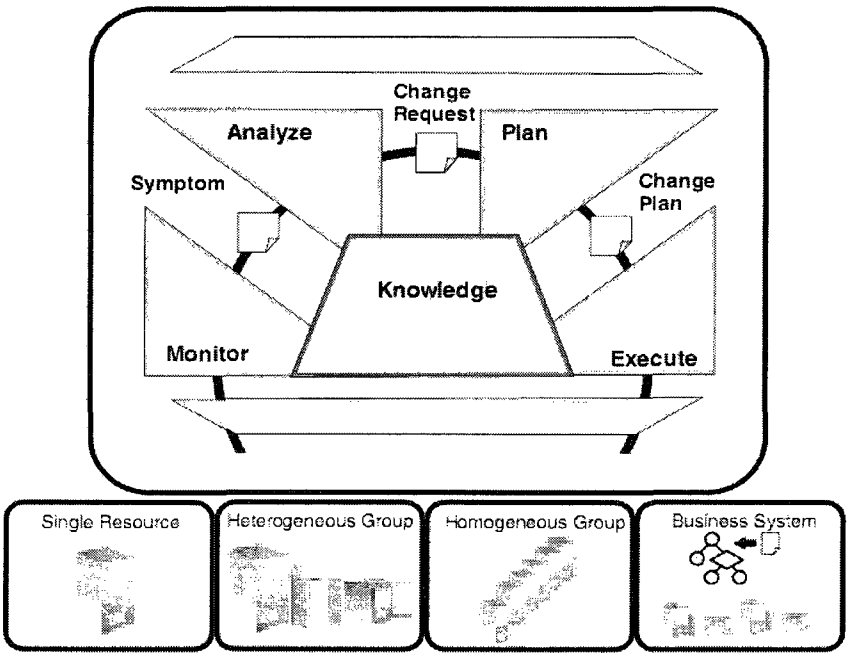

FIGURE 2.1: MAPE Loop

the analyze function triggers a change request in order to bring the resource back within its required parameters or in order to prevent the resource's functionality to go outside its required functional parameters. The change request causes the autonomic system to plan the changes required in order to restore or maintain the required functionality of the system. The change plan can be as simple as a single command or as complex as the execution of multiple workflows on different subcomponents of the resource. The planning function also deals with the possibility of resource conflicts, and attempts to generate a plan which does not break the functionality of other resources. Once a plan is generated, the execute function is responsible for the scheduling and execution of the changes required by the plan. The execute breaks the plan into its composing parts and ensures that the plan is executed in the correct order. The MAPE loop contains an extra component in the form of the knowledge repository. The repository's function is to provide the four functions the required information needed in order to run their logic. This could be information about how the resource behaves under various symptoms, what the required parameters are and possible solutions to maintain the parameters.

To illustrate an autonomic system and how the MAPE loop would be constructed

and work for it, the optimization of a cluster of servers will be taken as an example. The problem in many data centers is that the server clusters for various applications are set up for the worst case scenario. Outside of peak times, the server utilization is only 10 - 15 %, causing increased electricity bills for no reason. One way to solve the problem is to optimize the server usage, by starting and stoping servers as required by demand. As user demand increases the autonomic system will add server clones to the cluster, and as demand decreases some of the clones will be released back into a pool of free available servers. This allows the data center to have less servers than there are required by each of its applications' peak demands running at one time, as long as not all the applications peak at the same time. For this small example, it will be assumed that the application is a simple one tier web based application.

Based on the above example, the monitoring function would measure the response time, CPU utilization, and throughput of each of the servers, and then aggregate the data across all the servers in the cluster. Assuming a homogeneous cluster with Round Robin load balancing, a simple mean will provide a good estimate of the use of the cluster. The information regarding how to aggregate the data comes from the knowledge base which contains the topology of the cluster. These values are then fed forward to the analyzer function. The analyzer function takes the aggregated data, and compares it to known thresholds which represent a breach in the Service Level Agreement (SLA). In more complex cases the analyzer can approximate future values of the measured data and compare them to the thresholds. If a breach is detected, the analyzer triggers a change request which contains information regarding the size of the breach. The plan function takes the SLA breach information and creates a plan of how many servers to add or remove from the cluster. In complex cases the decision could include communicating with a top level manager to ensure that no conflict with other resources is caused by the plan. Once a plan is developed, the plan is sent to the executor. The executor uses the topology information in the knowledge base in order to decide which specific servers should be added from the pool or removed to the pool. Once the decision is made, the executor selects the appropriate workflows to execute on each server, and executes them. By doing this, the cluster can be optimized and can use less resources.

At the same time as the autonomic computing research got under way software companies like IBM, Sun Microsystems and Cisco all developed autonomic system

software solutions. An issue that appeared however is that the solutions provided by various software vendors do not collaborate and integrate well with each other. This is a problem if various autonomic systems from different vendors are deployed to manage separate parts of the IT infrastructure. Having the servers managed by one solution and the switches by another can cause problems if the two systems are incapable of interoperating. Since the IT infrastructure of most businesses is heterogeneous, it would be very possible to have such a mix of autonomic systems, especially if each of the low level resources has its own autonomic intelligence. Take for example a cluster of servers for a web application that uses a front end of IBM WebSphere Application Servers, with a back-end database composed of MySQL servers. If the autonomic intelligence for the application servers is provided by IBM as part of WebSphere and the autonomic management of the databases is provided by Sun Microsystems as part of the MySQL servers, then the two autonomic systems must be able to interact with each other. Without such an interaction each of the two systems would have an incorrect view of the full picture, and decisions made by each of them would result in non optimal results. For example, the WebSphere manager could decide to add application server clones to the cluster even though the actual bottleneck is the database cluster.

There is an extra problem which appears due to the fact that a common unified architecture is missing, and that is the need for different projects, both research and industrial, to rewrite code that was already developed by other parties. Furthermore, this redevelopment can cause research projects to spend valuable time developing components that are not part of the main research goals. While all the autonomic research tries to solve the same issues - those of self-configuration, self-healing, self-optimization and self-protection, the solutions proposed are varied ranging from machine learning algorithms to control systems. Various parts of the problem have also been attacked, varying from data gathering techniques to decision techniques. As each research project creates the entire MAPE loop this leads to inefficiencies. If the purpose of the research is to develop a new decision technique to self-optimize a cluster of servers, then it should be possible to pick up some Monitor and Analyze functions that provide the required parameters and work on developing just the Plan logic, while again not worrying about the Execute function. An extra benefit of using existing components, is that it becomes easier to compare just the new Plan function to previously existing solutions for planing and know that the other components do not cause any of the observed differences.

The above problems can be solved using a common unified architecture, which breaks the autonomic system into its composing parts. With such an architecture, a repository of components can be created, which work together to form an autonomic system. Thus researchers who want to explore and develop better components can simply use the available components for the parts of the system that they are not improving. At the same time, such an architecture would allow components from various vendors to interoperate, as the architecture would provide standard interfaces that must be exposed by different components.

2.2 Related Work

Autonomic computing research has been an important area of research in recent years with a number of publications and conferences supported by IEEE and ACM. The research has included varied disciplines and approaches in order to solve the problems, and various solutions for parts of the MAPE loop have been proposed. Solutions for the predictive part of an autonomic system include machine learning techniques and control loop techniques. The model of the autonomic system is also varied including queuing models, biological inspired models, policy based models and goal models. The change plan creation methodology also varies including decision tree based methods and algorithmic methods. While the proposed architecture must meet all the needs of various types of solutions, it must also be able to integrate with various vendor solutions by being based on open standards. This section will be composed of related work in autonomic system architectures, various approaches for the MAPE loop components, related software products and finally related open standards.

2.2.1 Autonomic system architectures

A number of approaches have been taken by different research groups in order to build the infrastructure needed for the creation of autonomic systems. This section looks at a number of other such architectures.

2.2.1.1 Autonomic Element based architecture

Work done at Montana State University [14] designs an autonomic system as an autonomic element by using object oriented approaches. Similar to the research in this thesis the goal of the paper is to design an architectural blueprint that can be used to quickly and simply develop other autonomic primitives. Also like this thesis, the paper starts from the MAPE loop designed by IBM, and goes into detail as to how the internals of each function should look like. The autonomic element, is seen in the paper as the base building block for an autonomic system. The element is composed of two parts: the autonomic manager and the managed element. In the paper, the element is described as a multi-threaded daemon, with each of the four MAPE functions as well as the effectors and sensors running in its own thread. Message passing is done via queues and synchronization is obtained via semaphores.

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.

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 [14] 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.

While this architecture has some common points with the subject of this thesis, there are also a number of differences between the two architectures. First of all, the autonomic element architecture executes its entire process on a single machine, while the architecture which will be proposed here is distributed across multiple nodes. Second of all, while the authors ensure the synchronization of the system and lack of deadlock in the system, the architecture does not deal with possible reliability problems in the execution of the control loop. Third of all, the architecture prevents the sending of messages backward and forward between components, something that, as will be shown later, can be desired in some cases. 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.

2.2.1.2 Self-managing peer-to-peer architecture

Another approach taken to build autonomic system is that of a self-managing peer-to-peer architecture, as described in [15]. 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,

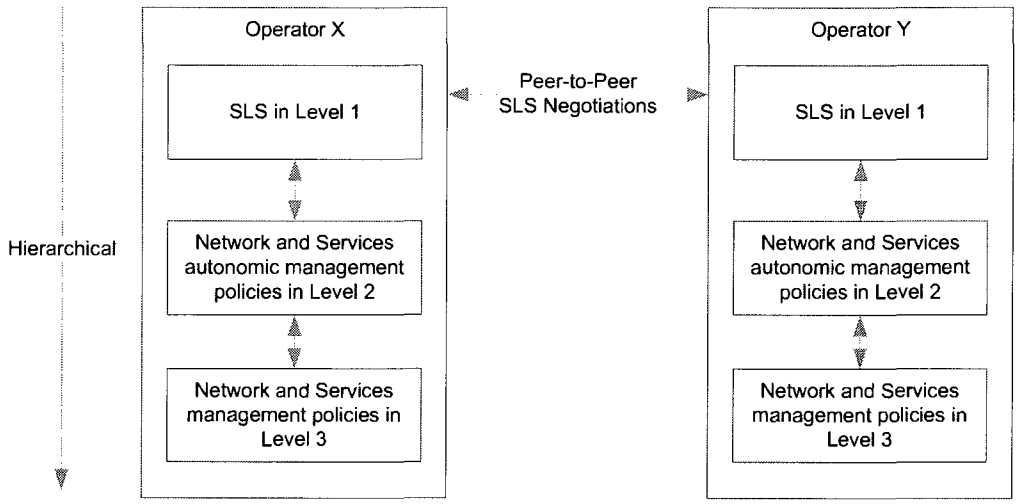

FIGURE 2.2: Self Managing Peer-to-Peer Architecture

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" [15], but it can also be extended to other autonomic systems. Figure 2.2, taken from [15], shows the behavior of the system.

For example, for a data center with multiple clusters of servers, the high level negotiation of how to split the existing servers in order for each cluster to reached it's required SLA can be done via peer-to-peer negotiations between the managers for the clusters. Once the negotiations are completed, each manager would pick up its own share of servers and set them up for its required goals. If at a later time, a cluster requires more server to maintain its SLA, it can again go into negotiation with other managers to try and receive more servers from those who do not need them.

The self-managing architecture in [15] does not provide any information regarding the implementation of the autonomic managers or elements, and is only of a very high level architecture describing how peer-to-peer can be used to achieve high level autonomic manager behavior.

2.2.1.3 Hierarchical control architecture

Another approach for creating autonomic systems is that of building a hierarchical control system, as proposed in [16] and [17]. In such a system, at the top, a data center controller monitors and manages the whole data center. At the middle level, each application deployed has its own controller which monitors the application, as well as receives configuration changes from the data center controller and its composing servers' controllers. At the bottom, each server has its own controller responsible for configuring and optimizing the server. If the server controller can not optimize the server by its own, it can request the application controller to try and optimize the whole application and its servers. If the application controller can not optimize the application and maintain its SLA and QoS, it can request the data center controller to take provisioning actions. In both architectures [16], [17] low level controllers can take very fast actions to optimize the system, but are restricted in how much they can affect the system. High level controllers can affect bigger parts of the system, but their provisioning actions require more time to complete and must use better estimates before taking any action.

Using the same example as before, of a data center with multiple clusters of servers, the low level server managers can make minor modifications regarding how the server behaves, like size of various pools, memory management, CPU management. The middle level application controllers, manage the entire application by making modifications regarding how many servers are being used, how many are free, how servers are distributed between various tiers, how the load balancing is being done, etc. Finally, the top level manager ensures that the entire data center behaves within the required SLAs and resolves possible conflicts. If an application that is very important requires more servers, than servers can be taken from a less important application for example. Figure 2.3 shows how the hierarchical architecture would look for a data center.

2.2.1.4 Biologically inspired and multi-agent architectures

A fourth architecture of autonomic systems is inspired by biological and multi-agent systems. In recent years, biologically inspired algorithms like the Ant algorithm [18] for best route discovery in a network have been investigated and developed. Similarly, for autonomic computing, biologically inspired architectures

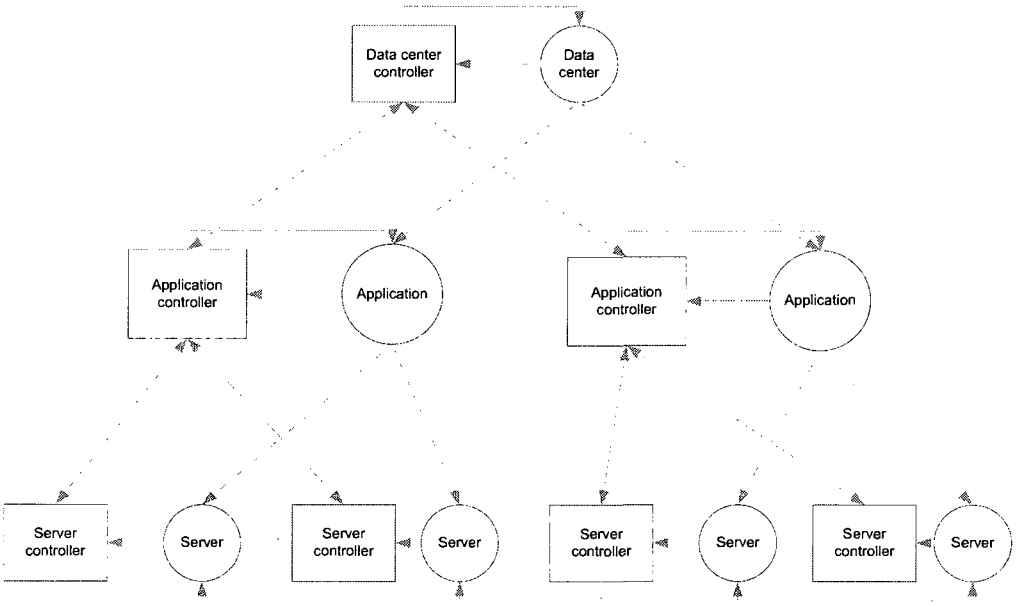

FIGURE 2.3: Hierarchical Architecture

have been investigated. In [19], 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. Figure 2.4 shows the design of a SelfLet.

A similar approach to that for SelfLets is presented in [20], where a system named Unity is introduced. Like the SelfLet concept, the creators of Unity develop the "self-management of a distributed computing system via interactions amongst a population of autonomous agents called autonomic elements". Unlike the SelfLet concept however, the high-level decisions of the autonomic system are obtained by using a resource arbiter, which is responsible for finding the optimum resource allocation. Furthermore, while in the SelfLet design the group to which a SelfLet

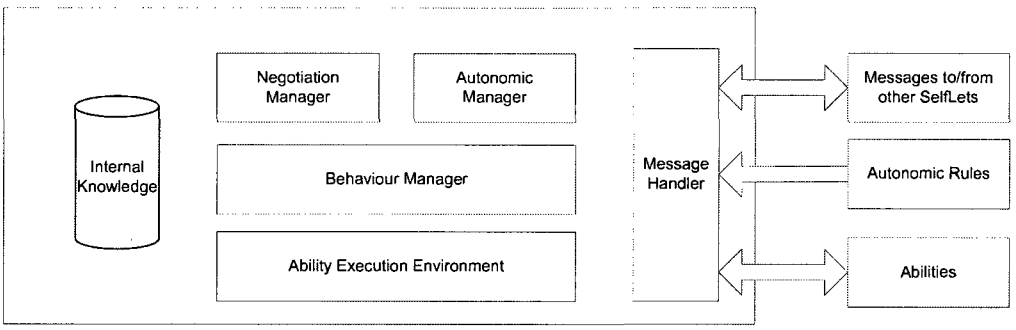

FIGURE 2.4: SelfLet Architecture

belongs is set beforehand by a human manager, in the Unity design autonomic elements group themselves based on their needs through the use of a registry component for discovery.

Due to the existence of a "global manager" in the form of the resource arbiter, the Unity design is closer to what an autonomic system tries to achieve. Without the existence of such a component it is impossible to achieve global optimums at the data center level, especially in cases where the demand is higher then the available resources and decisions of priority must be made. In the SelfLet design, each SelfLet tries to optimize itself to reach its own goals and without very careful design of policies instabilities in the system can be reached. Furthermore, the design of an autonomic element in Unity is very close to the MAPE loop design presented by IBM which is used for the development of the architecture presented in this thesis.

2.2.2 Autonomic system component research

While it is important to look at other architecture approaches taken to build autonomic systems, another important aspect to consider are the various methods used to build autonomic system components. Since the goal of the architecture designed in this thesis is to be usable by a large amount of researchers then it must be able to encapsulate different methodologies and concepts used by different solutions. Out of the four autonomic functions the analysis and the planning are the two that have the most varied types of solutions. Monitoring and execution are straightforward and do not result in different research approaches.

2.2.2.1 Machine Learning Analysis

One approach taken by researchers in order to analyze the performance of a system is by using machine learning algorithms, be it Bayesian networks or neural networks. The first step in analyzing a system and its behavior is to create a model of the system. The turning point for a correct design of an autonomic system is related to defining and building an accurate model of the system. The model provides a view of the controlled resource and both the future prediction of the state of the system and the creation of a plan to modify the system are based on the model. While a model can not perfectly represent the resource under control, it is important that the model accurately represent the state of the resource and more importantly accurately modify itself to changes in the environment. If the modifications in the system do not translate into correct modifications of the model, even an initial correct model will be out of tune with the modeled system after a number of iterations.

The complexity of deriving a model which can be used for solving at least the self-optimizing aspect is complicated due to a combination of types of processes such as computing, task scheduling, resource allocation solutions, and many others. In such a complex model, one can encounter discrete and continuous processes, discrete event systems, finite state machines, timed automata, and so on. On the other hand, there are two contradictory requirements for building good models:

1. the model has to be simple enough to facilitate design and
2. complex enough to contain the basic dynamics of the modeled process.

One very important feature of machine learning in general is that the system is capable of learning from a set of observed results, and then apply what it learned in future cases which might not perfectly match previously observed results. Thus a machine learning system is attempting to induce the full system behavior based on the way the system behaved in the past.

In [21] the authors compare the modeling of response time for a service-oriented architecture via Bayesian networks and neural networks. While the comparison results of the two methods is not important, what is important is how the two models are built and used. The case studied by the authors is a system composed of six services that work together to perform a certain user operation.

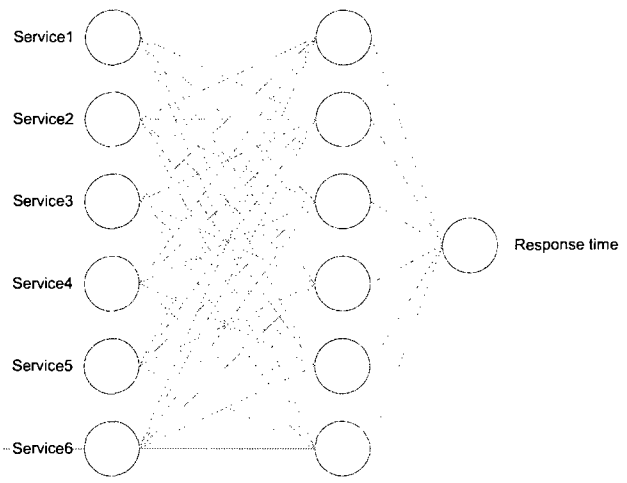

FIGURE 2.5: Neural Network Model for Response Time

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 modeling 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" [22]. In terms of the response time modeling, 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 a feed-forward neural network, with six input nodes, six neurons in the hidden layer, and one output layer with one neuron. 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. Figure 2.5 shows the design of the neural network.

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.

2.2.2.2 Queuing Model Analysis

A different approach for analyzing the behavior of computer systems is based on the principle of queuing models. Examples of such model design are presented in [23] and [24] where the authors develop models for multi-tier transactional applications based on Queuing Network Model (QNM) theory. The QNM looks at a system as a network of queues, including CPU queues, disk I/O queues, network queues, etc. Requests are split into various classes which represent different behavior of the modeled system when facing those requests. Any request of a certain class has to go through some of these queues, and the model of the performance (response time, throughput) is obtained by modeling the time a request waits in each of the queues, based on the workload. The structure of the queues can not be changed at run time and represents a static view of how the system must behave.

While the high-level modeling approach for the QNM is very similar to the Bayesian model described previously, the differences appear in how the model is built and used. The Bayesian model's parameters are chosen based on its update equations through observed values, while the QNM's parameters are computed through mathematical calculations before the system's deployment. At the same time, the Bayesian model's output is based on the previously known probability of an observed distribution, while the QNM's output is based only on the current size of the queues.

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. The approach taken in this thesis for modeling the response time of a cluster of servers is based on the queuing model approach, however the architecture proposed for autonomic systems can accommodate any of the described models.

2.2.2.3 Black Box Analysis

Another approach for analyzing autonomic systems is based on a black box approach. Unlike queuing models and machine learning models, which assume a certain structure to how the internals of the managed resource behave, black box

models are only considering the input variables and the output variables. No assumptions are made about the internals of the managed resource outside whether the resource behaves as a linear or non linear system. Based on observed input and output variables a parametric system of equations is then built using known control theory like Kalman equations [25]. Once the system of equations has been built, the best parameter values can be determined using known identification methods like recursive parameter estimation (RPE).

After the model has been fully determined, new input variables can be simply plugged in the equations and the predicted output variables can be found. The equations can be used to predict the future value of the output after a certain number of steps or time frames. The model equations can also be updated to better represent the system based on observed input/output relations.

Different systems of equations have been used in literature for building autonomic system models, as seen in [10] and [11]. One point in common however, is that computer systems are assumed to be non-linear since with the exception of the most basic, computer systems are not linear, and modeling them as such would result in poor performance.

2.2.2.4 Cost Function Planning

In an architecture based on the MAPE loop, once the analyze function has determined that a change must be done in the system due to a possible breach in the SLA or non optimal execution, a plan must be created in order to either maintain the SLA or try and reach an optimal execution. Like the analysis component, different plan function solutions have been proposed in literature. One such solution is based on cost functions and optimization of such functions. In [24] such a plan 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 functions 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.

Similarly in [26], the authors use a profit/cost approach combined with a policy approach to determine when the system is breaching the optimal execution and attempt to bring the system back to optimal execution.

2.2.2.5 Policy Based Planning

A different and simpler approach to creating a plan is based on policies. As shown in [27] and [28], policies can be used in order to determine how to optimize the managed resource based on the state of the resource. In the case of a data center with multiple server clusters that have conflicting requirements - for example demand is higher than the capacity of the data center - the system would decide how to distribute the servers based on the defined policy. For example the data center could have a policy that says:

if (cluster 1 demand is high) and (cluster 2 demand is high) and (has no free servers) then (cluster 1 receives 40% and cluster 2 receives 60%)

A major problem with policy based decision making is that policies have to be created by a human user before the system is deployed. At the same time, the policy creator has to consider all possible cases where the system must take a decision. On the other hand a policy based system has the advantage of having a fully defined behavior for a certain set of symptoms. Due to the need for policy creation, policy based planning is more appropriate in the case of a hierarchical architecture as a high level decision planner than as a low level decision planner.

2.2.2.6 Goal Model Planning

A third way to plan the actions required to optimize the system is through the use of goal models. Goal models are similar to policy based planning, however unlike policy planning which for a set of inputs has one possible action, goal models allow a certain number of actions for a set of symptoms. The system then picks the action that is the best action based on some internal metrics. In [29] the authors present how goal models can be used in order to create change plans for autonomic computing systems. A number of possible applications of the goal models are presented for self-configuring, self-optimization and self-healing. For example, for self-optimization the authors present an e-mail system where the client chooses one of three servers to connect to, based partially on server load.

The load function for each server is defined as a softgoal function, as described in following equation.

$$f(srv) = \begin{cases} \text{“}++\text{”} & ,if \quad load(srv) \leq 300 \\ \text{“}+\text{”} & ,if \quad 300 < load(srv) \leq 600 \\ \text{“}-\text{”} & ,if \quad 600 < load(srv) \leq 800 \\ \text{“}--\text{”} & ,if \quad 800 < load(srv) \leq 999 \end{cases} \tag{2.1}$$

The above function is only part of the goal based decision used to determine which server should be used by the e-mail client, and the functions have a certain contribution to the final decision. In goal modeling a (+) represents a help and a (-) represents a hurts relationship between goals and softgoals. Based on a computation of how its goals can be achieved based on the goal model and softgoal values the system decides which of the servers to use to achieve an optimal connection.

Like policy based planning, goal based planning requires the creation of the goals by a human user. Once created, the monitored values of the system are used to compute the softgoal values, which then combine to form a decision based on the goals of the system.

2.2.2.7 Machine Learning Planning

Another method used for planning changes in an autonomic system is based on machine learning classification and clustering techniques. In [30] a system named MESO is built for online decision making in autonomic systems. The system is applied to the problem of self-healing for an application which streams audio over the network. The streaming application described uses forward error correction (FEC) in order to fix some of the loses due to network packet loss at the receiver. The autonomic system's goal is to decide how much FEC to use based on the network's loss rate. As the loss rate increases, more FEC is used, and when the loss decreases no FEC or little FEC is used.

In order to learn the appropriate FEC values, the authors train the system by randomly dropping packets and using reinforcement learning. A human user selects a FEC configuration for the observed packet loss, and if the perceived loss is under an acceptable level then the configuration is reinforced. Based on multiple learning instances, the system learns a clustering approach which matches

the input/output values given. Once the system is in use, based on the observed packet loss at the receiver, a request is made to the sender for a specific FEC configuration which minimizes the bandwidth/information loss.

While the authors train the system using human input, methods to automatically cluster available data exist and can be used to build such an autonomic planner.

2.2.2.8 Control Theoretic Planning

Due to being well-established in most engineering fields and being backed by a well understood theoretical background, which includes stability analysis and accuracy analysis, control theoretic approaches have also been proposed for the MAPE loop's plan function. Control theoretic approaches as shown in [31] base themselves on a black box modeling approach, where the state of the system is defined by some linear or non-linear equations. As such, finding the planning action that must be taken in order to reach the desired state requires optimizing the given set of equations.

In [31] the authors examine the use of a control theoretic approach in order to optimize the response of a database which is exposed as a Web Service. When the response size for a query to the database is large, the response must be broken into blocks in order to avoid out of memory problems. The autonomic part of the system must decide what block size to use in order to optimize the response time. Based on test observations as the block size increases, the response time decreases up to a certain value, and after that as the block size increases the response time decreases. The goal of the system as such, is to find the optimal value of the block size which results in the minimum response time.

For the self-optimization system that will be presented later in the thesis, a control theoretic planing function will be used. Like the modeling of the system, all the various planning functions have a number of similarities which can be modeled in the architecture. First of all, all the plan functions are based on the current observed inputs of the system and the modeled state of the system. Second of all, the plan functions try and calculate an optimum value for the state of the system under control. Finally the plan functions compute the required actions to go from the current state to the optimum state.

2.2.3 Related software

As mentioned previously, a number of software companies have developed autonomic computing systems. At the same time, autonomic systems manage other software systems. Since one of the goals of the architecture presented in this thesis is to integrate easily with existing software systems, both autonomic and managed systems, it is important to look at and understand what systems which already exist do.

2.2.3.1 Related autonomic systems

Two of the first autonomic systems were the IBM Tivoli Provisioning Manager (TPM) and Tivoli Intelligent Orchestrator (TIO), which is based on TPM [32]. TPM is a simple deployment engine and is unable to analyze a system and determine what action to take. The goal of TPM is to receive a request for an action to take and decide on a plan which would execute the required action. For example, TPM can receive a request to add extra servers to cluster. In the face of such a request TPM would find available servers and based on predefined workflows provision those servers and add them to the cluster. Unlike TPM, TIO contains an intelligent component, which 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 uses TPM, which is part of the TIO product, to deploy additional servers. A number of issues exist with the TIO/TPM 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. 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.

Another autonomic computing solution, for web application management, is Sun's N1 Service Provisioning System (SPS) [33]. 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 TPM, SPS is capable of provisioning operating system clones and applications in order to quickly configure a new server. Like TIO and TPM, Sun's N1 system is a closed proprietary system which is hard to extend.

A similar product to N1 SPS and TPM is Enigmatec's Virtual Orchestrator (EVO) [34]. EVO is geared specifically for virtual machines and permits the quick deployment of virtual machines for a certain request for resources. Based on predefined policies EVO provisions servers, deploys the appropriate virtual machines and once the lease time expires releases the servers back into the available pool.

While the above software systems are all closed source software, there also exist some open source autonomic software systems. Scalr [35] is a self-optimizing, self-healing autonomic system for Amazon's Elastic Cloud Computing (EC2) grid. Scalr allows users to create farms of servers on EC2 and then is capable, in the case of failure, to provision new servers and synchronize data between servers. Scalr also provides some basic self-optimization capabilities in that it monitors the load average across the server's in a farm and when the load average goes above or bellow certain thresholds the system adds or removes servers from the farm.

The above systems all represent starting points for autonomic systems, however none of them offer an open architecture, which would allow users to build their own intelligent components. At the same time, the capabilities offered fall short of what a full autonomic computing system needs to offer in terms of intelligence.

2.2.4 Related Open Standards

One of the major requirements of autonomic systems according to IBM's description is that autonomic systems must "function in a heterogeneous and open way, and implement open standards" [4]. Since the final goal of autonomic systems is to automatically manage a given set of resources, open standards in management are especially important. A number of open standards for resource management and resource instrumentation exist.

2.2.4.1 Application Response Measurement (ARM)

The Open Group's Application Response Measurement (ARM) [36] is an open standard for enterprise application instrumentation. The standard defines APIs which are used to monitor the execution of an application from the point of view of units of work which are important for the application. The API defines ways for an application to inform a monitoring agent when a transaction starts and when a transaction ends. The decision of what a transaction represents is left to the application that is being monitored. The agent also receives from the application information regarding the completion status of the transaction, and context data like parent/child relations between transactions. The agent receives the data, transforms the data into appropriate measurements and passes the measurements to a management application.

Due to the parent/child relations that are created between transactions, ARM allows the measurement of end-to-end call trees. The ARM standard can be used in autonomic systems to provide a unique interface for all manageable resources which care about transactions time. However, in an autonomic system transaction times are not the only important metric to measure, and as such for more complex measurements a different standard has to be used.

2.2.4.2 Web Service Distributed Management (WSDM) and WS-Management

WSDM [37] and WS-Management [38] started as competing standards that had the same final goal - creating a standard for management of resources through the use of web services. Recently the parties involved with each of the two standards, OASIS and DMTF respectively, have started working on creating a single set of standards for management of resources via web services that would reconcile the two sets of standards.

Both standards define, each in their own slightly different way, the idea of a resource. Resources are manageable entities, which provide a certain number of capabilities through which a managing application can gather data from the resource and also modify the resource. All communications are done via SOAP and each resource has its own unique endpoint where it can receive messages. Both standards also define ways to create notification based systems, where managers

can subscribe to resources and receive events from these resources. The standards also define a way to exchange meta data between a resource and a manager regarding the capabilities of the resource. An extra feature of WSDM is the definition of the idea of a metric. Metrics are measurements that a resource can provide to a manager which represent the state of the running resource.

The two standards do not define what kind of data can be measured or what type of resources can be managed. The standards define the messaging formats only and it is up to resource developers what data they want to provide to outside systems. It is important to note that software vendors have already started supporting WSDM and WS-Management. IBM provides a feature pack for WebSphere which adds WSDM management capabilities for some of its data. Similarly Microsoft includes Windows Remote Management in Windows Vista and Windows Server 2008, which is based on WS-Management. Other vendors like Novell, Sun, Oracle have also developed or are working on developing management capabilities based on one of the above two standards for their products. The final goal is that all similar products, like databases or application servers would provide the same data in the same format. For example, all application servers would provide the response time for a servlet in a similar manner, such that an autonomic manager can transparently manage application servers and not be restricted to only WebSphere servers or Weblogic servers.

The framework presented in this thesis is based on the WSDM standard due, in part, to the fact that it provides support for metrics.

2.3 Architecture Requirements

The first step in building the architecture for any software system, regardless of the development paradigm being used, is developing the requirements of the system. Only once the requirements of the system are well understood can an architecture which meets the requirements be built. As such, the following requirements were determined for the architecture presented in this thesis based on past research, existing systems and the IBM autonomic computing manifesto.

Componentized In order to build an architecture which contains standard interfaces and functionality, and which can be reused in various self-management

aspects the system must be split into separate components. At the same time each component should perform only one role in the system resulting in high cohesion. Such a system can use various types of models, controllers, and decision makers and the components can interoperate no matter what type of analysis or planning is used. This split will result in an ability to mix and match components to create autonomic systems.

Distributed The components can be distributed across multiple nodes and component to component messaging goes through the network. This provides higher reliability as a hardware failure only results in part of the system's failure and the components that failed can be moved to a different node.

Minimal coupling The communication between components must happen via clearly specified interfaces in order to minimize the coupling between components. These interfaces define the contracts between components similar to Service Oriented Architecture (SOA) interfaces. Since the architecture is distributed over multiple nodes no other messages are allowed. The interfaces exposed by each component will represent the functionality of a component

Push based messaging The communication between the components must be push based, with components sending the data when it is available. This also results in high cohesion within the components, low coupling, and allows multiple recipients for the same data. Since the data is sent when it becomes available, no extra time is spent polling the other components. Due to the distributed nature of the system, a polling based system would also result in higher network utilization and higher response times.

Reconfigurable The control loop itself must be reconfigurable at run time, either through user actions, self-reconfiguration or reconfiguration request from some other component in the environment. This creates a management system that is able to modify itself based on its execution and environment. Components in the autonomic control loop can be stopped/started/removed/created/changed on demand, and while they are modified, if possible, the rest of the control loop continues to function.

Reliable and fault tolerant The autonomic solution itself must be fault tolerant and reliable. Since the final goal is to build a system which can manage other resources by itself, some of the resources being critical, the system

must be capable of having very high uptime and must provide guarantees regarding its decisions.

Based on open standards As defined in the manifesto an autonomic system must work with components from various software and hardware providers, and as such it must use and be based on open standards.

The high level design of an architecture, rooted in real time software engineering patterns, developed using the Model Driven Development (MDD) practices and based on open standards, is presented in Chapter 4.

Chapter 3

Model and Identification of Autonomic Computing Systems

As showed in Figure 2.1 of Chapter 2 and in more details in Chapter 4 the autonomic computing structure maps well in a classical control loop technology.

In this thesis the control loop technology will be exploited to build an interoperable architecture for the Autonomic Computing Infrastructure such that control loop elements and laws are interchangeable. Under this design concept, the system itself is able to make decisions in regards to the changes desired and proceed with appropriate actions towards this goal.

The control technique, which will be used in this thesis, follows the modern approach of state estimation and robust control which has its foundations in [39]. According to this approach, whatever the plant is, a state estimator will provide the proper information about the evolution of the states inside the plant infrastructure. At the same time, a Kalman gain will assure that the controller controls the system such that the system error due to stochastic variations in the system variables will be minimized and eliminated.

This elegant strategy, which revolutionized the whole domain of system and control theory and applications, requires the existence of apriori knowledge of the model of the system. This in turn requires the analysis of all physical or domain laws which govern the evolution of the processes taking place in the plant (in our case the clusters on which the application servers are executed) and combines them in a state space model in order to describe the plant from a theoretical point of view.

As the Kalman filter cannot work with arbitrary parameters (symbols) there is a need to know numerically which are the values of the parameters used in the theoretical model above. As such an identification process has to take place. The identification process is nothing else than the algorithm used to find which are the numerical values of the parameters of the system and at what order the algorithm has to stop.

To comply with the above requirements, in the present Chapter an abstract model of the autonomic computing processes will be sought. This model will then be identified in order to obtain the best numbers for the system parameters and the order of the matrices used. In an intermediate step, a special model and parameter estimator for the Kalman filter will be considered, such that it will cover certain nonlinearities present in the model used which will imitate the real nonlinear behavior of the plant.

3.1 Queuing and Scheduling Model

A very important decision when building an autonomic computing system consists in modeling the process that will be controlled. Based on the model created, a system which updates the model using observed data can be chosen and finally a control law can be also built. This section presents the background theory used when building the model and control law for the response time of a cluster of application servers which are running a multi-tiered application.

In what follows, the dynamic processes taking place in a JEE server will be analysed and a model for these dynamic processes will then be built. It is considered that a JEE application server receives a request from a client for a servlet, performs some processing, creates one or more database requests and finally builds a response which it sends back to the client. At the same time an application server uses a thread pool which contains a number of threads. Each of these threads can process requests. Once all the threads in the pool are being used, new requests are added in a queue which uses a First Come First Served (FCFS) policy while they await a request under process to complete and a thread in the pool to become available. After the request has been processed, the corresponding reply is added to a queue in order to be sent back to the client. Figure 3.1 shows the strucuture of the considered model.

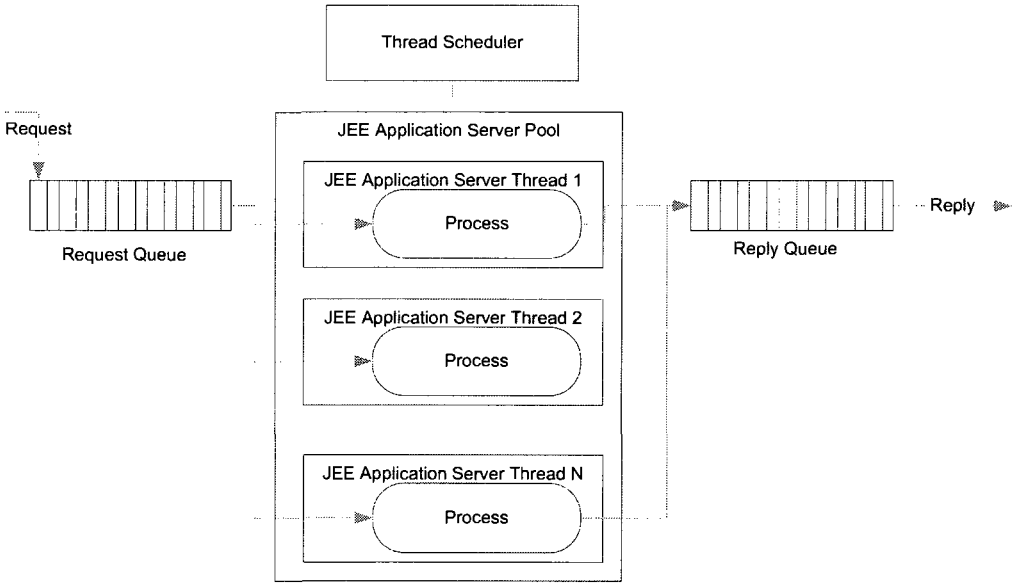

FIGURE 3.1: Application Server Request Model

Assuming the following notations:

1. $rt(k)$ is the response time for request k in milliseconds, with $rt(k) \in \mathbb{R}_{\geq 0}$ and $k \in \mathbb{N}$
2. $rt_{thread}(k)$ is the response time for request k in milliseconds due to being processed by a thread, with $rt_{thread}(k) \in \mathbb{R}_{\geq 0}$ and $k \in \mathbb{N}$
3. $rt_{buff}(k)$ is the response time for request k in milliseconds due to buffer queuing, with $rt_{buff}(k) \in \mathbb{R}_{\geq 0}$ and $k \in \mathbb{N}$
4. rt_{avg} is the average response time for all requests
5. $ct(k)$ is the computing time for request k in milliseconds, with $ct(k) \in \mathbb{R}_{\geq 0}$
6. $bt(k)$ is the I/O waiting time for request k in milliseconds, with $bt(k) \in \mathbb{R}_{\geq 0}$
7. $st(k)$ is the time process k waits to be scheduled in milliseconds, with $st(k) \in \mathbb{R}_{\geq 0}$
8. N represents the total number of requests scheduled while a certain request is waiting to be scheduled. $N \in \mathbb{N}$
9. $q(k)$ is the size of the buffer queues when request k arrives, measured in number of requests, with $q(k) \in \mathbb{N}$

10. $U(k)$ is the number of incoming requests expressed as a piece-wise constant, non-decreasing, over the time interval $[0, k]$ with $U(k)$ from $\mathbb{R} \rightarrow \mathbb{N}$

11. $V(k)$ is the number of the outgoing requests expressed as a piece-wise constant, non-decreasing, over the time interval $[0, k]$ with $V(k)$ from $\mathbb{R} \rightarrow \mathbb{N}$

12. $u(k)$ and $v(k)$ are the time derivatives of $U(k)$ and $V(k)$ respectively

13. $r(k)$ is the rate of processed requests, with $r(k) \in \mathbb{N}$

14. a, b, c, d are all constant values for a system, with $a, b, c, d \in \mathbb{R}$

15. $resp_t$ is a function representing the average response time of the system, at time t, with $resp_t \in \mathbb{R}_{\geq 0}$

16. $cpuLoad_t$ is a function representing the CPU load of the system, at time t, with $cpuLoad_t \in \mathbb{N}$ and $0 < cpuLoad_t < 100$

17. $arrivalRate_t$ is a function representing the arrival rate of request, at time t, with $arrivalRate_t \in \mathbb{N}$

18. $buffState_t$ is a function representing the state of the application server buffer, at time t, with $buffState_t \in \mathbb{N}$

19. $Sat_Q(q)$ is a multivaried function representing the saturation of the buffer, when the buffer contains q requests, with $Sat_Q(q) \in \mathbb{N}$

The model for the response time for an application server can be decomposed in two parts as in Equation 3.1: a scheduling part and a queuing part. The scheduling part represents the time it takes for a request to be processed by a server thread, from the time the request is passed to one of the threads in the thread pool, to the time a response is sent to the client. Even though an application server uses multiple threads to process requests, the underlying operating system and hardware will limit the number of threads being run concurrently. However, the processing of a server request will not run uninterrupted as data from a database will be required to build the reply. When a request becomes blocked due to database I/O, other threads can execute their own requests. The queuing part of the response time represents the time the request waits in a buffer for a free thread in the thread pool.

The response time due to scheduling can be modeled using the same methodology as in real-time systems [40]. In the simplest form, where there is only one processing core and only one database request per user request, the response time for a request due to scheduling can be expressed as in Equation 3.2:

$$rt(k) = rt_{thread}(k) + rt_{buff}(k) \tag{3.1}$$

$$rt_{thread}(k) = ct(k) + bt(k) + st(k) \tag{3.2}$$

For modeling purposes ct_k and bt_k can be assumed to be constants for a given request type, since they represent the exeuction time of a request and the time a request is blocked due to database I/O. st_k however depends on how the other threads are released for processing and how long they take to process. This can be expressed mathematically as follows:

$$st(k) = \sum_{j=0}^{N} ct(j) \tag{3.3}$$

where $ct(j)$ represents the time it takes to process a request j which is scheduled before request k. This results in a total response time for scheduling as in equation 3.4.

$$rt_{thread}(k) = ct(k) + bt(k) + \sum_{j=0}^{N} ct(j) \tag{3.4}$$

However, as noted before, a request does not wait only to be scheduled, it also waits in a buffer for a thread in the thread pool to become available. This situation can be modeled using principles from network buffer modeling and more specifically using a fluid flow model [41], [42]. In the fluid flow model a buffer is seen as being controlled by the following law:

$$q(k+1) = q(0) + U(k) - V(k) \tag{3.5}$$

which upon derivation with respect to times, as shown in [41], becomes:

$$q(k+1) = q(k) + u(k) - v(k) \tag{3.6}$$

However, any buffer has a size limit. Despite the fact that applications servers and operating systems have a large capacity for processing requests, there are situations when the server slows down considerably. Once the buffer is full, any incoming requests are discarded. In the case of an application server, this will result in client timeouts. The model for response time must deal with this case also. As such, similar to the work done in [42], the buffer state can be defined as:

$$q(k+1) = Sat_Q \{q(k) + u(k) - v(k)\} \tag{3.7}$$

where

$$Sat_Q(q) = \begin{cases} 0 & if q \leq 0 \\ Q & if q \geq Q \\ q & otherwise \end{cases} \tag{3.8}$$

The response time, due to the queue wait, is then a function of the size of the buffer when the request is received. As such, the response time for request k due to buffering can be approximated as the buffer size at time k when the request arrives multiplied by the average response time for a request in the previous time frame $k-1$.

$$rt_{buff}(k+1) = q(k) * rt(k) \tag{3.9}$$

The outgoing number of requests themselves can be calculated based on the processing rate of the application server, which will be denoted $r(t)$. The final expression for the response time assuming an empty initial processing queue, thus becomes:

$$rt(k+1) = ct(k+1) + bt(k+1) + \sum_{j=0}^{N} ct_j + Sat_Q \{q(k) + u(k) - r(k)\} * rt(k) \tag{3.10}$$

In order to determine how a system under such a model will behave in the future, the control system must be able to predict ahead of time the state of the buffer queue's capacity as well as of the scheduling system. The scheduling system itself, depends on the load of the processing core the application server is running on. The buffer queue's capacity depends on the arrival rate, and the processing core which affects $r(k)$ and $R(k)$, thus the response time can be expressed as a function:

$$resp_{k+1} = a + b * cpuLoad_k + buffState_k * resp_k \tag{3.11}$$

and

$$buffState_{k+1} = Sat_Q \{buffState_k + c * arrivalRate_k - d * cpuLoad_k\} \tag{3.12}$$

with

$$resp_{k+1} \rightarrow \infty \quad when \quad Sat_Q(q) = Q \tag{3.13}$$

where the average response time at time $k+1$ is a function composed of a constant factor - which represents the average computation and I/O wait times for a request, the *cpuLoad* function at time k which represents the state of the scheduling system, and the response time at time k multiplied by the state of the queues which depends on arrival rate, CPU load and previous buffer state.

3.2 Extended Kalman Filter

The model previously built is a state-space model, and the equations for expressing the response time are non linear equations. In order to estimate the future value of the average response time, based on the current average response time, arrival rate and CPU load, an Extended Kalman Filter (EKF) can be used [25]. The Kalman Filter was developed originally in order to predict the output of a linear dynamic system, based on an approximation of the state of the system. Kalman Filters are developed as Markov chains built on top of linear systems with Gaussian noise. At

each time step, the Kalman Filter computes the new state based on the previous state and the measured inputs. Once the new state is calculated, the output of the system is predicted from the new computed state. This allows the Kalman Filter to require only the state for the previous step. No history is required, as the state of the system is evolved with each computation.

Due to how useful the Kalman Filter proved to be for the control of linear systems, the Extended Kalman Filter was developed in order to solve the same problem for systems which have non-linear state transitions or observation models. This is achieved by linearizing the non-linear functions around the state estimate at the time of prediction.

The following notations will be used for the Kalman Filter formulas:

1. x_k is the state vector of the system, $x_k \in \mathbb{R}^n$.
2. z_k is the measured data vector for the system, $z_k \in \mathbb{R}^m$.
3. u_k is the driving function of the system, $u_k \in \mathbb{R}^l$.
4. $f(x, u)$ is a non-linear function which relates the previous state and the driving function at step k with the state at step $k+1$. This function can be expressed as follows:

$$x_{k+1} = f(x_k, u_k) \tag{3.14}$$

5. $h(x)$ is a non-linear function which relates the state x_k to the measurement z_k The function can be expressed as:

$$z_k = h(x_k) \tag{3.15}$$

6. $\hat{x}_k$ represents an *a posteriori* estimate of the state performed at a previous step k
7. $\tilde{x}_k$ and $\tilde{z}_k$ are the approximations of x_k and z_k respectively
8. w_k represents the process noise, as a random variable
9. v_k represents the measurement noise, as a random variable
10. A is the Jacobian matrix of partial derivatives of f with respect to x

11. W is the Jacobian matrix of partial derivatives of f with respect to w
12. H is the Jacobian matrix of partial derivatives of h with respect to x
13. V is the Jacobian matrix of partial derivatives of h with respect to v

For the model described in the previous section, the Extended Kalman Filter structures would translate as follows:

The state vector of the system is composed of three elements - response time, request buffer size and CPU load. This is expressed in vector form as:

$$x_k = \begin{bmatrix} cpuLoad \\ responseTime \\ bufferSize \end{bmatrix} \tag{3.16}$$

The measured data is represented by the CPU load. This is expressed in vector form as:

$$z_k = \begin{bmatrix} cpuLoad \end{bmatrix} \tag{3.17}$$

The driving function is the arrival rate of requests, expressed as:

$$u_k = \begin{bmatrix} arrivalRate \end{bmatrix} \tag{3.18}$$

$f(x, u)$ is a non-linear function which relates the CPU load, response time and buffer size at time $k+1$ to the CPU load, response time and buffer size at time k and to the arrival rate at time k. This function can be expressed as follows:

$$\begin{bmatrix} cpuLoad \\ responseTime \\ bufferSize \end{bmatrix} = \begin{bmatrix} cpuLoad + a * arrivalRate \\ b + c * cpuLoad + bufferSize * responseTime \\ Sat_Q \{bufferSize + d * arrivalRate - e * cpuLoad\} \end{bmatrix} \tag{3.19}$$

$h(x)$ is a function which relates the measured CPU load to the state of the system formed by the CPU load, response time and buffer size. The function can be expressed as:

$$\left[\ cpuLoad\ \right] = \left[\ cpuLoad\ \right] \tag{3.20}$$

The filter can be expressed mathematically via the following equations:

$$\tilde{x}_k = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.21}$$

$$\tilde{z}_k = h(\tilde{x}_k, 0) \tag{3.22}$$

which can be linearized as follows around the value of the state at time $k-1$:

$$x_k \approx \tilde{x}_k + A(x_{k-1} - \hat{x}_{k-1}) + Ww_{k-1} \tag{3.23}$$

$$z_k \approx \tilde{z}_k + H(x_k - \tilde{x}_k) + Vv_k \tag{3.24}$$

with A, W, V and H defined as follows:

$$A_{[i,j]} = \frac{\partial f_{[i]}}{\partial x_{[j]}}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.25}$$

$$W_{[i,j]} = \frac{\partial f_{[i]}}{\partial w_{[j]}}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.26}$$

$$H_{[i,j]} = \frac{\partial h_{[i]}}{\partial x_{[j]}}(\tilde{x}_k, 0) \tag{3.27}$$

$$V_{[i,j]} = \frac{\partial h_{[i]}}{\partial v_{[j]}}(\tilde{x}_k, 0) \tag{3.28}$$

Like any other Kalman Filter, the computations of the Extender Kalman Filter are done in two steps, with Q and R representing the process noise covariance and measurement noise covariance respectively.

1. Time update, which is the prediction stage. At this stage the Kalman Filter predicts the future state of the system and the covariance matrix for the estimate using the following two equations:

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.29}$$

$$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \tag{3.30}$$

2. Measurement update, which is the Kalman Filter correction stage. At this stage the filter corrects its internal structure based on the measured values. This is done by first computing the Kalman gain, followed by updating the state and covariance matrix as shown in the next three equations:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \tag{3.31}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, 0)) \tag{3.32}$$

$$P_k = (I - K_k H_k) P_k^- \tag{3.33}$$

For the model described previously the A, W, V and H matrices can be defined as follows:

$$\begin{aligned} A_{[i,j]} &= \frac{\partial f_{[i]}}{\partial x_{[j]}}(\hat{x}_{k-1}, u_{k-1}, 0) \\ &= \begin{bmatrix} 1 & c & -e \\ 0 & bufferSize & 0 \\ 0 & responseTime & 1 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} H_{[i,j]} &= \frac{\partial h_{[i]}}{\partial x_{[j]}}(\tilde{x}_k, 0) \\ &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \end{aligned}$$

with W and V being identity matrices.

3.3 Model Identification

Once the model has been built for a system, the model's parameters must be identified. The problem of initializing the model in the case of the Extended Kalman Filter presented previously, consists in finding the initial values of the matrices that compose the model, as well as the required sampling rate. This is done by applying model identification techniques. Model identification techniques

FIGURE 3.2: Black Box View

work by assuming that the system is a black box, with certain inputs and outputs. As presented previously, the input for the model of a server is the arrival rate of clients and the outputs of the system are the response time and the CPU load of the server, as shown in figure 3.2.

A black box model system can be transformed from the time domain, where the output function is the convolution of the input and the impulse functions, into the frequency domain, where the output function is the product of the input function and the transfer function. The transfer function is the frequency domain equivalent of the impulse function. The transformation from the time domain into the frequency domain can be obtained via a Laplace transformation of the functions.

In order to determine the sampling interval, a stochastic input is applied to the black box and the output values of the system are measured. The autocorrelation function is applied to the input and output values, and then the Discrete Fourier transform of the autocorrelation functions for the input and output is taken. Through this method, the transfer function, defined as:

$$\frac{F_{output}}{F_{input}} = |J_{transfer}|^2 \tag{3.34}$$

can be found, where F_{output} is the Fourier transform of the autocorrelation of the output values and F_{input} is the Fourier transform of the autocorrelation of the input values, while $J_{transfer}$ is the transfer function.

The graph of the transfer function's magnitude versus frequency is taken, and the intersection with the x axis of this graph represents the frequency of the signal, which translates directly into the sampling timing required for the input variables.

The second problem that model identification deals with, is that of calculating the initial values of the matrices that compose the model. In the EKF model presented previously this consists in finding the initial values of the A and H

matrices. This can be achieved by sampling a number of input/output pairs and using a Recursive Prediction Error (RPE) approach. Like in the case of sampling determination, a stochastic input is fed into the system, and the output is then measured. At its base an RPE algorithm deals with the development of a recursive formula to estimate the parameters of the form:

$$\hat{\Phi}(t) = \hat{\Phi}(t-1) + \mathcal{L}(t)Z(t) \tag{3.35}$$

where $\Phi(t)$ is the vector containing the parameter estimates at time t, $\mathcal{L}(t)$ is the gain for the parameter estimate and $Z(t)$ is the observation error, defined as the observed output ($z(t)$) minus the predicted output ($\hat{z}(t)$), or:

$$Z(t) = z(t) - \hat{z}(t) \tag{3.36}$$

In order to build a model for parameter estimation a criterion [43] of the following form must be minimized.

$$N(\Phi) = \epsilon[\frac{1}{2}Z^T(t)\Pi^{-1}Z(t)] \tag{3.37}$$

where Π is a positively defined matrix, for which, as shown in [43], the best choice is the true prediction error covariance matrix. The minimization of the criterion is achieved by doing iterative approximations [43], via for example a stochastic Newton method. The Newton method is generalized via successive approximations of the form:

$$f(t+1) = f(t) + j\frac{g(t)}{\frac{d}{dt}g(t)} \tag{3.38}$$

where $f(t)$ is the solution, $g(t)$ is the function being solved and j is the gain factor. Applying the Newton method to solve for the optimal solution of $N(\Phi)$, which means solving $\frac{dN(\Phi)}{d\Phi} = 0$ results in an equation of the form:

$$\hat{\Phi}(t) = \hat{\Phi}(t-1) - \mathcal{V}(t)\left[\frac{d^2}{d\Phi^2}N(\Phi)\right]^{-1}\left[\frac{d}{d\Phi}N\Phi\right]^T \tag{3.39}$$

$$= \hat{\Phi}(t-1) - \mathcal{V}(t)\left[\frac{d^2}{d\Phi^2}N(\Phi)\right]^{-1}\Delta(t-1)\Pi^{-1}Z(t-1) \tag{3.40}$$

where $\Delta(t)$ is the gradient of the observation error, and $\mathcal{V}(t)$ is a time decaying gain factor.

The second derivative of the criterion can be approximated recursively as follows:

$$\mathcal{R}(t) \approx \frac{d^2}{d\Phi^2}N(\Phi) \tag{3.41}$$

$$\mathcal{R}(t) = \mathcal{R}(t-1) + \mathcal{V}(t)\left[\Delta(t)\Pi^{-1}\Delta^T(t) - \mathcal{R}(t-1)\right] \tag{3.42}$$

The gradient of the observation error can be derived from the Kalman equations as:

$$\Delta(\Phi, t) = \left[\frac{d}{d\Phi}z^-(\Phi, t)\right]^T \tag{3.43}$$

$$= \left[\frac{d}{d\Phi}H_k x^-(\Phi, t)\right]^T \tag{3.44}$$

$$\Delta(\Phi, t) = \left[H_k\frac{d}{d\Phi}x^-\right]^T \tag{3.45}$$

The problem is this approach is that the estimate of second derivative is very sensitive to round-off errors, and as such a more stable approximation of the derivative must be used. In order to achieve this a sequence $\lambda(t)$, called forgetting factor, is defined, as follows:

$$\lambda(t) = \frac{\mathcal{V}(t-1)}{\mathcal{V}(t)}\left[1 - \mathcal{V}(t)\right] \tag{3.46}$$

Based on this sequence, a new factor $\mathcal{P}(t)$ is defined as:

$$\mathcal{P}(t) = \mathcal{V}(t)\mathcal{R}^{-1}(t) \tag{3.47}$$

and a form of the gain function can be defined as follows in order to be more numerically stable:

$$\mathcal{L}(t) = \mathcal{P}(t-1)\Delta(t)\left[\Delta^T(t)\mathcal{P}(t-1)\Delta(t) + \lambda(t)R_z(t)\right]^{-1} \tag{3.48}$$

$$\mathcal{P}(t) = \left[I - \mathcal{L}(t)\Delta^T(t)\right]\mathcal{P}(t-1)\left[I - \Delta(t)\mathcal{L}^T(t)\right]/\lambda(t) + \mathcal{L}(t)R_z(t)\mathcal{L}^T(t) \tag{3.49}$$

$$\Phi(t) = [\Phi(t-1) + \mathcal{L}(t)Z(t)]_{\mathcal{DM}} \tag{3.50}$$

where

$$R_z(t) = H_k P_k^- H_k^T + V_k R_k V_k^T \tag{3.51}$$

and $[]_{\mathcal{DM}}$ represents a mapping function such that the parameter vector is also stable. A common way of doing this, is by multiplying $\mathcal{L}(t)$ with a weighting factor that decreases from 1 to 0, and selecting the factor that results in a stable system. The stability can be tested by ensuring that all the eigen values are within the unit circle [43].

It is necessary to define a sequence for the forgetting factor $\lambda(t)$. This should have a low value initially and approach a value near 1 when the data is processed in order to achieve a stable result. Like in [43], the forgetting factor can be defined as:

$$\lambda_i = \lambda_{ss} + \lambda_{rate}(\lambda_{i-1} - \lambda_{ss}) \tag{3.52}$$

where λ_{ss} is the steady state value of λ, and λ_{rate} determines the rate at which $\lambda(t)$ approaches λ_{ss}.

All the above equations can be given in terms of the model developed earlier for the Kalman Filter to form the RPE algorithm as follows:

$$\lambda_i = \lambda_{ss} + \lambda_{rate}(\lambda_{i-1} - \lambda_{ss}) \tag{3.53}$$

$$\Delta_k^T(\Phi) = H_k \frac{d}{d\Phi} x^- \tag{3.54}$$

$$\mathcal{S}_k = \Delta_k^T(\Phi)\mathcal{P}_{k-1}\Delta k(\Phi) + \lambda_k R_{z(k)} \tag{3.55}$$

$$\mathcal{L}_k = \mathcal{P}_{k-1}\Delta_k(\Phi)\mathcal{S}_k^{-1} \tag{3.56}$$

$$\mathcal{P}_k = \frac{1}{\lambda_k}(I - \mathcal{L}_k\Delta_k^T(\Phi))\mathcal{P}_{k-1}(I - \mathcal{L}_k\Delta_k^T(\Phi))^T + \mathcal{L}_k R_{z(k)}\mathcal{L}_k^T \tag{3.57}$$

$$\Phi_k = [\Phi_{k-1} + \mathcal{L}_k Z_k]_{\mathcal{DM}} \tag{3.58}$$

$$\mathcal{K}_k = \frac{d}{d\Phi} K_k(\Phi) \tag{3.59}$$

$$\Sigma_{a(k)}(\Phi) = \frac{d}{d\Phi}\hat{x}_k \tag{3.60}$$

$$= (I - K_k(\Phi)H_k) + \mathcal{K}_k Z_k(\Phi) \tag{3.61}$$

$$\Sigma_{b(k+1)}(\Phi) = \frac{d}{d\Phi} x_k^- \tag{3.62}$$

$$= A(\Phi)\Sigma_{a(k)}(\Phi) + \frac{dA(\Phi)}{d\Phi} x_{k-1} \tag{3.63}$$

The initial conditions required for the parameter estimation are Φ_0, $\mathcal{R}_0$, λ_0, λ_{rate}, λ_{ss} and $\Sigma_a(0)$

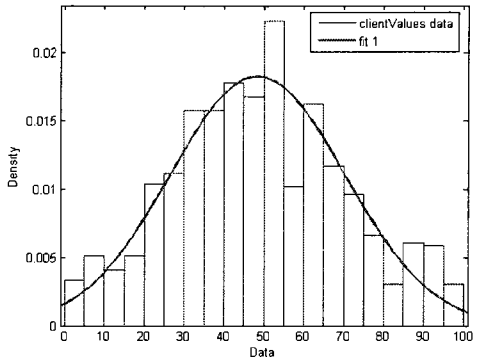

FIGURE 3.3: Client distribution

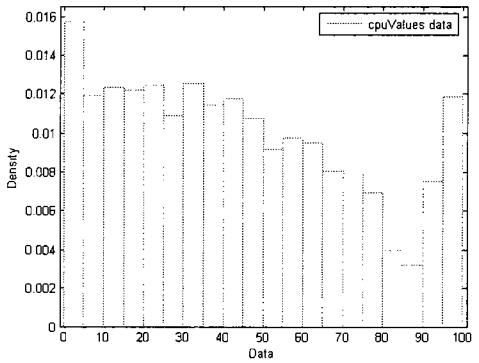

FIGURE 3.4: CPU Utilization distribution

3.3.1 Model Identification Results

In order to determine the sampling rate, the arrival rate of clients - represented by a client distribution - was created using a Gaussian distribution with the following parameters:

$$\mu = 50, \sigma = 25 \tag{3.64}$$

The number of clients accessing the server was changed every minute, with data for number of clients gathered every 30 seconds, and the data from the server was gathered every 10 seconds. Figure 3.3 shows the client distribution for the experiment for 787 data points, which is equivalent to the experiment running for six hours and a half. Figure 3.4 shows the CPU utilization distribution, and figure 3.5 shows the response time distribution.

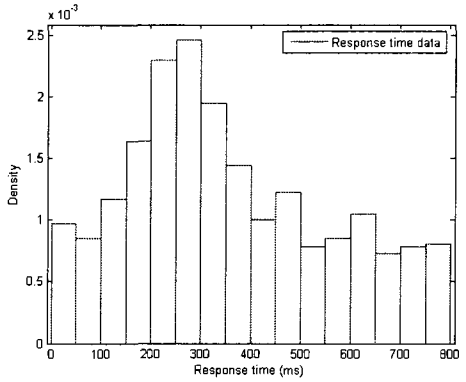

FIGURE 3.5: Response time distribution

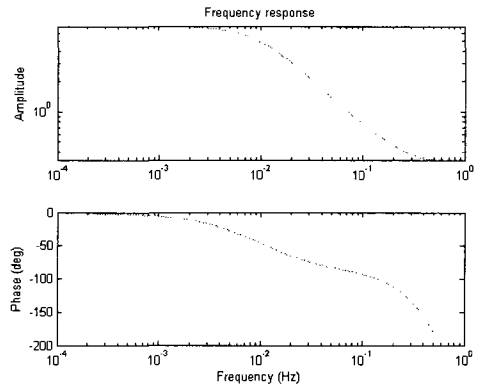

FIGURE 3.6: Magnitude and Phase of Transfer Function

By taking the Fourier transform of the autocorrelation functions for the input and output and diving them, the transfer function can be obtained. If the magnitude and phase of the transfer function versus the frequency are plotted in a Log-Log plot, the required sampling rate for the system can be found. The frequency at which the magnitude is zero is equal to the required sampling frequency, and the sampling rate is simply:

$$SampleRate = \frac{1}{Frequency} \tag{3.65}$$

Figure 3.6 shows the magnitude and phase of the response function. By zooming into the graph it can be seen that a magnitude of 0 is obtained for a frequency of approximately $10^{-1.14}$Hz, which means in this case a sampling rate of approximately 13.80 seconds. Since this is an approximation, and the sampling rate

should be less then the found sampling rate, the sampling rate used of 10 seconds is satisfactory.

By applying the RPE method described previously to the same data set used for determining the sampling interval, the following initial matrices were found for the control of the application server:

$$A_{[i,j]} = \begin{bmatrix} 0.68797 & 0.61823 & -0.079551 \\ 0.34731 & 0.88649 & -0.55752 \\ -0.22688 & 0.83512 & 0.54463 \end{bmatrix}$$

$$W_{[i,j]} = \begin{bmatrix} 0.0026355 \\ 0.0034263 \\ 0.0056881 \end{bmatrix}$$

$$H_{[i,j]} = \begin{bmatrix} 784.76 & 209.64 & -355.31 \end{bmatrix}$$

$$V_{[i,j]} = \begin{bmatrix} 0 \end{bmatrix}$$

and

$$K_{[i,j]} = \begin{bmatrix} 0.00045276 \\ 0.00060938 \\ 0.001044 \end{bmatrix}$$

This model results in a fit of the data of 18.58% as shown in Figure 3.7. For the purpose of this thesis, this will provide good results for the model - in part due to the iterative nature of the EKF. More research must be done however in applying various types of nonlinear identification methods like the ones in [44] or [45].

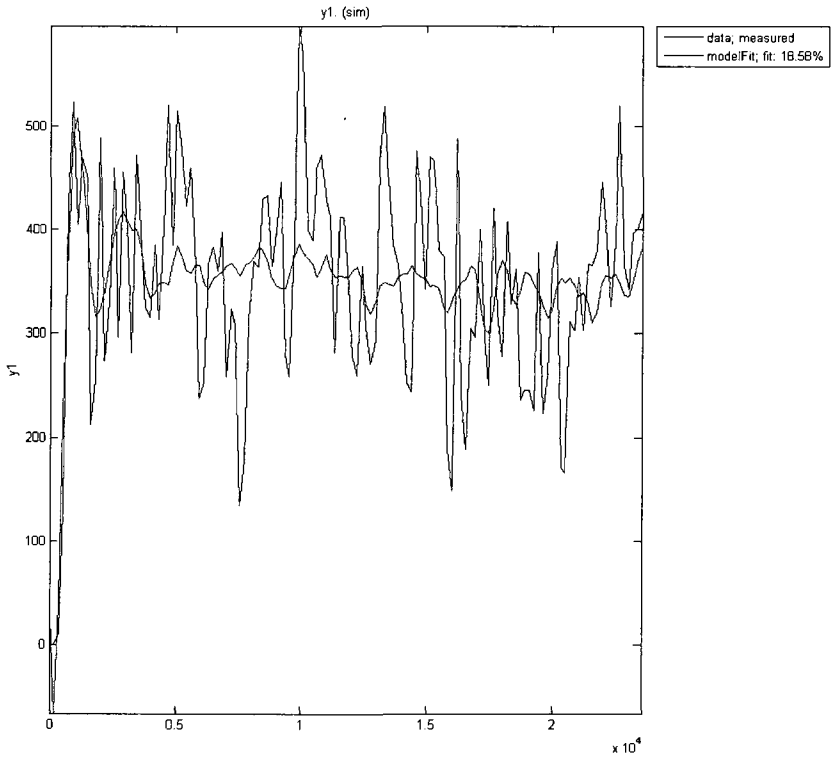

FIGURE 3.7: Data and Model Fit

Chapter 4

High Level Design

The development of the architecture, and of the framework built based on the architecture uses the Model Driven Development paradigm. In the MDD paradigm the first step is to build a Platform Independent Model (PIM). The PIM represents the architecture of the system in a platform agnostic way, without making any reference to any language of development, system on which to execute or method of communication between components. As such the PIM represents a very high-level view of how the system is composed and how various components interoperate with each other. For the architecture presented in this thesis the PIM defines the components that form an autonomic system, the functionality that each component provides as well as the common functionality for all the components. The second step in developing a system using MDD is the development of the Platform Specific Model (PSM) which will be presented in Chapter 5.

The starting point of the architecture is the MAPE loop as defined by IBM in [4]. In the MAPE loop shown in figure 2.1, the functionality of an autonomic system is split into four subcomponents: the *monitor* function which gathers data from the managed resource, the *analyze* function which takes the measured data and analyzes it in order to determine the state of the system, the *plan* function which creates a plan to modify the resource based on analyzed data, and the *execute* function which performs the necessary modifications to the resource. The problem with the MAPE loop, is that some of the four functions perform more than one functionality. The analyze function models the system, predicts the present and future states of the resource and determines any breach of policy. At the same time, the plan function uses a model of the resource to determine how

to modify the resource's state and creates a plan that achieves the modification. Furthermore, in some cases the data provided by the resource is not the same with the data expected by other components, in which case either the monitor or the analyze function will have to modify that data. For example, for a cluster of servers each server will provide its own CPU utilization but the analyze function analyzes the whole cluster and is interested in the average across all servers. In this case the analyze function will need to average the utilizations across all servers.

4.1 Component Design

One of the main goals of the architecture is to allow the reuse of components created by separate entities with minimal effort. In order to achieve this, object composition is used throughout the whole design of the system. By building an autonomic computing system via object composition, the system can obtain the appropriate functionality dynamically at runtime by choosing the appropriate components for the task and composing them into a full autonomic computing environment. Furthermore, this generates a system that is capable of modifying the structure at runtime, by simply replacing a component in the system with another component with equivalent functionality. The composability of the system is achieved by enforcing that communications between components are done only via the component interfaces. This results in minimal coupling between components, and high cohesion within components.

The design of each of the components takes the composability one step further however. Each of the components, which will be described later, is created by combining low level blocks that perform specific functions. These functional blocks can be shared between components of different types. For example, the notification based block, which will also be described later, can be added to any of the components by simply creating an object of the right type and communicating with it via the object's interfaces. The component creator is shielded from knowing the internals of the notification mechanism and can only subscribe to some types of notifications or publish notifications, depending on whether the system added the notification client or producer block respectively. Note that this does not mean that a component must choose between a producer and a client. A component is free to compose both of them and become both a client and a producer for notifications.

4.2 Component Descriptions

Because of the problems with the MAPE loop, the architecture splits the autonomic system into eight high-level components. Each of these components have only one clearly defined role, and do not perform any function outside the given role. All the interfaces for the components define ways to access the data or the functionality of the components. An autonomic system can be obtained by composing some or all of these eight components. An autonomic system can also contain more than one of a type of component. The eight components of the architecture are:

Sensors The sensor performs the monitor function in the MAPE loop. Its single goal is to gather raw data from the managed resource. The sensor does not process the data gathered, and only makes it available to the next component in the system. As such the sensor must interface with the underlying resource, retrieve the necessary data and format the data to the required output. For example, WebSphere's Java Management eXtension (JMX) module provides many statistic values since the server started. For the purpose of reusability, the sensor that monitors WebSphere server data, provides to the rest of the system the values since the server started. This allows the use of the same sensor in a system that requires a value since the server started, or a system that requires a value just for the last thirty seconds.

At the same time, all of a sensor's data must come from the same source. A WebSphere Application Server sensor can not gather data from the operating system or the hardware. The goal of this requirement is allowing reuse of sensors. In order to alleviate the need for measuring data from separate sources a single resource can have multiple sensors, each of them providing different data from the resource. For the same example as before, a WebSphere based server can have a sensor that monitors the WebSphere software execution, a sensor that monitors the CPU usage at the operating system level and a sensor that monitors network traffic at the network card level. By having the sensors separated, the same CPU sensor can be used for other deployments that use the same OS, and the same network sensor can be used for other machines with the same network card. Furthermore, this allows for easy composition of sensors at deployment. By knowing the

hardware and software in a system, a manager can easily choose and deploy from a repository sensors that measure data from the system.

Filters In the MAPE loop, filters would reside either in the monitor or the analyze function. The filter's role is to take data coming from sensors and modify it based on the requirements of other components. The filter communicates with the sensors in order to retrieve data available in the sensors, processes this data based on some internal structure and makes the data available to other components in the control loop. For the WebSphere server example, the filter would be able to compute the statistic for the last thirty second interval, based on two consecutive measured observations. Because of this, the filter must be able to store data that it receives from a sensor previously and use it at a later time.

The filter can also aggregate data across multiple sensors, thus forming a many to many structure between sensors and filters. For a cluster of servers, a filter can gather data from each of the servers regarding the CPU utilization, and then generate a unique view for the whole cluster based on statistical values like average, mean, minimum, and maximum CPU usage. In some cases it is also necessary for the filter to compute new measurements that are not directly available from the managed resource but that are necessary later in the control loop. For example, an autonomic system that requires the throughput of a server, but the server can not provide the throughput as a measurement. Throughput being defined here as the rate of successful operations, be it in *requests/second*, *bytes/second*, etc.. In such a case a filter can gather data from the server regarding the number of successful operation together with the timestamp when the number of successful operations was observed and compute throughput from the two values.

In order to achieve complex behavior from simple components, each of the components performing one filtering role, filters can be chained together. First a filter can compute the values for the last thirty seconds. Based on the results for the last thirty seconds, the throughout for those thirty seconds can be computed by a second filter in the chain. Similar to the sensors, this allows for the reuse of various filtering techniques in multiple control loops.

Model The model acts as part of the knowledge component in the MAPE loop. It is used by both the analyze and the plan functions in order to predict and to optimize the state of the system. A valid and stable model is paramount

for an autonomic system. The model must be able to represent the state of the system accurately enough for decisions to be made regarding changes in the system. The model must also be able to maintain a correct view of the system as the system and its environment change. As presented earlier, the model can be created using any of a number of different methods ranging from control theoretic models to machine learning based models. The goal of the model is to hold the autonomic's view of the managed resource. Since in autonomic computing the managed resource is a complicated, non deterministic system with different processes, task scheduling, resource allocations, the model will represent just a portion of the state of the system. As such the model offers its modeled variables to other components which require it for their own computations.

At the same time, computer systems are not static and the model must be capable of updating itself based on newly observed measurements in order to preserve a good model of the resource. This can include retraining the model or recomputing the parametric values in the model based on known equations. Because of this, the model can receive update requests from other components, together with measured or computed data that is the basis for the update. The knowledge of how the model is to be updated is part of the model however. The model can be seen as a black box by the other components of the system.

Estimator The estimator acts as part of the analyze function in the MAPE loop. Its goal is to use the modeled data and any new measured data in order to estimate the future state of the system. Like the model, the estimator can use various approaches to determine the future state like machine learning methods or control theoretic methods. Since the decision of making modifications to the system is based on estimated data, the estimator must provide stable estimates in the face of changing environments. The estimator allows other components to request a new estimate of the system's state at a future time T based on some input variables, and through the use of the model the estimator predicts and returns the state at time T.

The estimator and the model interact frequently since the estimates are based on the current known state of the system, as presented by the model. Furthermore the estimator can trigger changes in the model based on estimated data in order to keep the model in line with the system's state. In

order to achieve this, the estimator and the model must be able to have a common knowledge of what the model's parameters represent.

Coordinator The coordinator is the brain of the autonomic system. Its function is too coordinate the execution of the other components in order to achieve the final goal of the autonomic system. It is responsible for gathering data from the filters and using this data and its own internal logic to decide the order in which to execute the other components. It is the coordinator who decides when to estimate the future state and how far ahead to estimate and it is also the coordinator who decides when to create a change plan. As such, the coordinator can communicate with all other components of the system with the exception of the sensors and adaptors.

In order to understand why a coordinator is required an example is presented next. For some control theoretic approaches to autonomic computing, like Kalman filters for example, it is necessary to compute the estimate of the future state multiple times. As new estimates are computed, the values of those estimates will approach one another. A valid estimate is one obtained when two consecutive values are within a certain threshold. It is the coordinator who decides, in such cases, when to execute the estimate and how many times to execute it. Other control methods like machine learning, do not require multiple estimates and the coordinator's driving logic would be different.

Decision maker The decision maker is the component which generates a change plan. Based on a request coming from the coordinator, which asks for a decision to be made regarding necessary changes in the system, the decision maker uses the model's data, its own internal logic as well as preset knowledge in the form of QoS and SLA levels in order to generate a decision which maintains the managed resource's SLA and QoS. Like the estimator, the decision maker uses the state of the system as presented by the model in order to reach its plan.

Logically, the decision maker will implement an optimization algorithm which attempts to find the optimal value which keeps the system within its SLA. It is extremely important that the decision maker is stable and does not cause fluctuations in the manged resource since some modifications can take minutes to complete. For example, for a cluster of servers the addition of a new server to the cluster can take upwards of five minutes. In such cases it

is important that on one hand the decision maker does not attempt to add servers to the cluster multiple times for the same symptoms and on the other hand does not add and remove servers from the cluster frequently. While the decision maker can access the state of the system through the model, unlike the estimator it can not make changes in the model.

Actuator The actuator is equivalent to the execute function in the MAPE loop. It is responsible for executing a change plan created by the decision maker. This can include scheduling actions for a later date, communicating with various components of the resource and requesting changes, and executing workflows. The actuator also ensures that in the case of a failure in the execution, the action is rescheduled for a later time if possible. Because of this, the actuator must be capable of not only sending a request to the resource but also monitoring the execution of that request. For example, an actuator that has to add a new server to a cluster needs to perform two high level actions - first it must install a server clone on the new machine, and once that is done it must notify the cluster controller about the new server. The new server installation can be done by executing remote commands on the new server via SSH for example. In this case the actuator would monitor the execution of the remote commands and ensure that all the commands finish successfully.

The actuator must have an extra capability in order to roll back unsuccessful changes. If a change request is being executed and it fails before successfully completing, the actuator must be able to roll back any changes made. For the example above, if the execution fails while the cluster controller is trying to add the new server to its internal structure, the newly created server would go back to the available pool after the new software is uninstalled.

Adaptor The adaptor does not exist as a component in the MAPE loop, but it is necessary in order to ensure that the control loop works correctly. Its responsibility is to adapt the rest of the autonomic system to changes in the managed resource. Unlike the model which makes changes based on measured data and estimated data, the adaptor makes changes based on structural changes to the managed resource. For example, for the cluster example given above, when a new server is added to the cluster the adaptor ensures that the correct sensors are deployed on the new server, that the

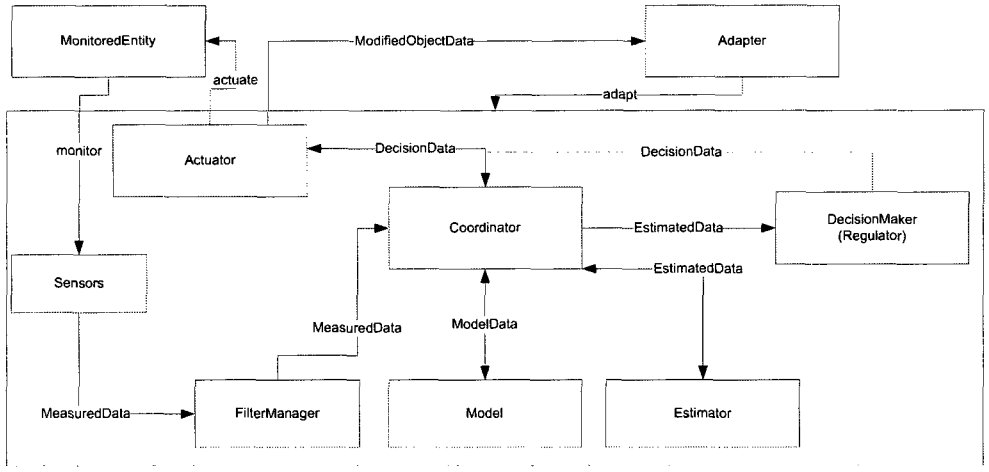

FIGURE 4.1: Autonomic Control Loop

filters know about the new sensor and that the model is updated with the new number of servers.

The adaptor can also be used to make structural changes to the control loop if there are changes in the environment. Such changes can include completely replacing one or more components if the state of the environment changes sufficiently that they can no longer operate correctly and control the managed resource correctly. Because of this, the adaptor must be able to send control commands to all the other autonomic system components and also receive notifications regarding changes in the environment and the managed resource.

Figure 4.1 shows the high-level architecture of the autonomic system, based on the eight components described previously. While the architecture defines the eight components, a control loop does not need to implement all the eight components if they do not make sense logically. For example a control loop that uses raw data for its analysis and decision can use the data directly from the sensors and not use a filter.

4.3 Message Structures

In order to ensure the correct communication between components a number of common message objects are created that define the structure of the messages passed between various components.

Measured Data The *Measured Data* structure defines the message structure of the messages between the sensors and filters and between the filters and the rest of the control loop. Such a data structure can contain multiple measured data entities in the same message. Each of the entities is composed by the following fields, with each field represented by a name/value pair.

1. Name, representing a unique ID which identifies the measured data
2. Value, representing the value of the measured entity
3. Data type, representing the type of the value - string, integer, double, etc.

Model data The *Model Data* structure defines the message structure which represents the internal data being held by the model. Like the *Measured Data* data structure can contain multiple subentities in the same message. The structure of these entities is:

1. Name, representing a unique ID which identifies the model data
2. Value, representing the value of the model entity
3. Data type, representing the type of the value - string, integer, double, etc.

Estimated data The *Estimated Data* structure is used to pass data from the estimator to the coordinator and from the coordinator to the decision maker. This data represents the future predicted values of the estimator. This data structure contains the following:

1. Name, representing a unique ID which identifies the estimated data
2. Value, representing the value of the estimated entity
3. Data type, representing the type of the value - string, integer, double, etc.

Decision data The *Decision Data* structure is used to pass data from the decision maker to the coordinator and from the coordinator to the actuator. This data structure represents the result of a decision made by the decision maker and is used to instruct the actuator what actions it needs to take. This data structure contains the following:

1. Name, representing a unique ID which identifies the decision data

2. Value, representing the value of the decision entity
3. Data type, representing the type of the value - string, integer, double, etc.

Modified Object data The *Modified Object* data structure is used to pass data from the actuator to the adaptor. This data structure contains information regarding the results of the actuation and changes made in the managed resource. It tells the adaptor how the managed resource changed, such that it can adapt the control loop. One data structure can contain multiple entities, one for each part of the managed resource which changed. Each entity contains an array of name/value pairs:

1. Name, representing a unique ID which identifies the entity's property
2. Value, representing the value of the entity's property
3. Data type, representing the type of the value - string, integer, double, etc.

While all the data structures have the same common format, they are separated in order to allow developers to extend one of them if needed, and to split the logical meaning of the messages being passed in the system.

4.4 Autonomic Computing Registry

The architecture presented in this thesis has an extra component, which is not part of the autonomic computing control loop, in the form of the autonomic computing registry. The registry has two main goals: to allow system administrators to create autonomic control loops by choosing from the autonomic components available, and to allow human operators to monitor the state of a previously created autonomic system. In order to achieve these goals the autonomic registry provides three interfaces to clients:

Register interface The register interface allows components, autonomic systems and manageable resources to register or modify themselves. Autonomic system components register themselves on startup and remove themselves when they shutdown. Autonomic systems get registered when they are created by

a system administrator. Manageable resources are added to the registry by an administrator or the system can scan the network for possible resources. Even in the case of scan, an administrator is required to choose which entities are manageable and which are not. The information in the registry is backed in a persistent storage.

Retrieve interface The retrieve interface is used to query data from the registry. A client can connect to the registry and perform a query to obtain data regarding available components, autonomic systems or manageable resources. Components can be queried by location (server node where it is deployed) , type (sensor, filter, etc.), or name. Autonomic system's can be queried by a unique name given at creation or by type of managed resource. Manageable resources can be queried by type.

Remove interface The remove interface is used to remove data from the repository. Components and manageable resources can remove themselves at shutdown, autonomic systems can be removed when an administrator stops the system and destroys it. The removal is done based on a unique ID of the resource. Objects removed through this interface are removed from the persistent storage.

The registry component also defines the structure of the objects that can be persisted - components, systems and manageable resources.

Manageable Resources The *Manageable Resource* data structure stores information related to system resources that can be managed by autonomic systems. Each manageable object contains the following fields:

1. Name - a unique identifier of the resource
2. Description - a human readable description of the resource
3. Location - how the object can be accessed, like a URI
4. Type - type of the resource like server, database, router, etc.
5. Object type - the object type can be one of
   (a) "non-manageable" - which represents an object that can not be directly managed like a multi-tier application

(b) "manageable" - which represents an object that can be directly managed like a cluster of servers

(c) "endpoint" - which represents information about objects that can be directly modified in order to impact the performance of a manageable object like the servers in a cluster.

6. List of children objects - servers in a cluster
7. Link to its parent object if one exists and was persisted in a previous request

Autonomic component An *Autonomic Component* data structure stores information about either a component which is available to be deployed as part of an autonomic system, or one which is already deployed as part of a running autonomic system. The component data structure contains:

1. Name - a unique identifier for the component in its container. Different components can have the same ID, as long as they are deployed in different containers - where container is defined by a combination of the machine and process in which the component is residing.
2. Description - a human readable description of what the component does
3. Type - which can be one of the eight component types described before
4. Location - URI where the component can be accessed, this will be different between an available component and a running component
5. Status - the executing status of the component, can be one of: available, configured, running, paused, stopped

System The *System* data structure represents an already created and deployed autonomic system. The fields of the system data structure are:

1. Name of the autonomic system - unique identifier given at creation time
2. Status of the autonomic system - whether it is running, stopped, paused or reconfiguring
3. Process type - what kind of process is being automated by this autonomic system be it a server cluster, or a virtual machine cluster, etc.
4. Links to *Manageable Object* data structures - which represent the objects which are managed by the autonomic system

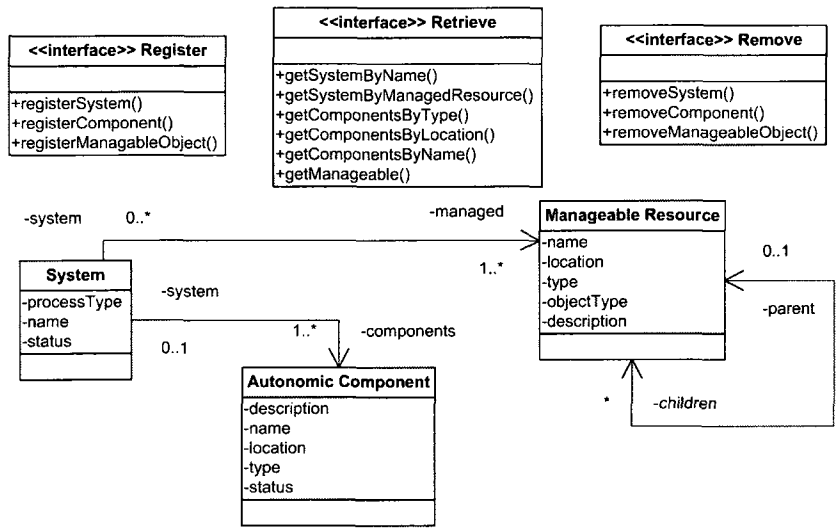

FIGURE 4.2: Registry Structure

5. Links to *Autonomic Component* data structures - which represent the objects which form the autonomic system.

Through the use of the registry, a client can create an autonomic system by choosing the appropriate components and managed entities based on those available in the environment. Once created the system is persisted in the registry and a client can use the information in the registry to connect to the components and monitor their execution. Figure 4.2 shows the structure of the registry's interfaces as well as the data structures and the relation between data structures.

4.5 Real-time patterns

An autonomic system must itself be reliable and fault tolerant. In order for system administrators and IT managers to trust automation in data centers and in the IT infrastructure, they must be able to trust the autonomic solution to perform the correct actions even in the face of incorrect stimuli, and to recover from possible failures in such a way that the infrastructure remains managed at all times. The first part of this requirement means that the data is constantly being validated as it passes through the control loop in order to ensure that incorrect actions are not being taken. For example, a request to remove more servers than are currently available from a cluster would result in no action being taken, as the validation

unit for the actuator would detect an incorrect request and refuse to perform it. The second part of this requirement means that the autonomic manager is capable of saving its state and in the case of complete or partial failure, is able to recover and restart itself from a known safe state. For example, if the model fails while the control loop is executing the autonomic system is capable of restarting the model based on the previous saved state. Furthermore, the rest of the control loop does not fail just because the model fails, and is intelligent enough to pause itself until the model is back online and running.

Another issue caused by the distributed nature of the control loop is that of message synchronization. Since each component can run on a different node, and messages are passed through the network as notifications, issues such as latency and jitter can impact how the control loop is performing, especially if the control law requires high frequency calculations. Jitter is more of a problem than latency though, as jitter can result in multiple messages being delivered at once, or even being delivered out of order. Depending on the type of component, this can result in incorrect calculations. For example, in the case of a control loop that does iterative estimations, and which updates the model based on each estimate, an incorrect order of messages could result in incorrect updates to the model. In addition, the use of SOAP over HTTP (which uses a reliable transmission protocol - TCP/IP) for data communications can result in message retransmission in the case where messages are lost in the network. Message retransmission can cause the same problems as jitter - incorrect order of messages.

Both of the above types of problems can be solved however using principles and patterns from real-time systems, as reliability, synchronization and fault tolerance are problems real-time systems always deal with. Because of this, the thesis infuses a number of real-time patterns into the architecture to deal with these problems.

4.5.1 Reliability

In order to deal with the problem of reliability, the architecture uses an instance of the protected single channel pattern. This pattern adds validation checks at specific points in the control loop's communication channel. At each of these points, the message either coming into a component, or going out of a component is checked and validated to ensure consistency. This pattern results in less fault tolerance than a redundancy pattern, but at the same time it is not as expensive

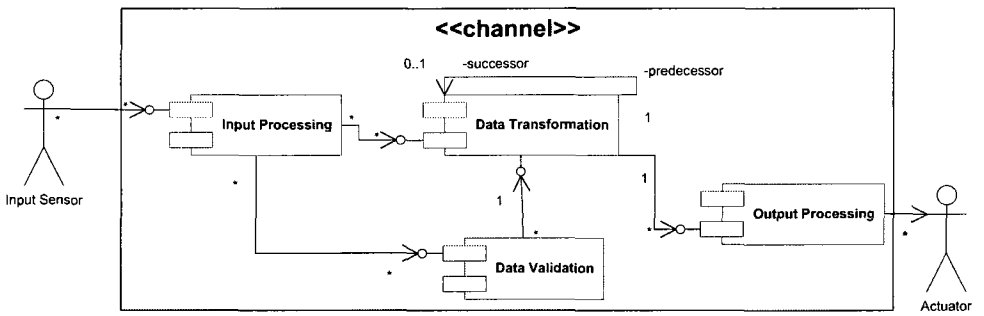

FIGURE 4.3: Protected Single Channel Pattern

resource-wise as a pattern that uses multiple control loops and a voting system to choose which action to use. A simple example is a sensor measuring the response time of an application server - if the number of clients requesting a certain servlet is high, but the response time is zero, this would suggest that there is a problem with the measurements and that the data either should not be passed to the filter if it is validated at the output of the sensor, or should not be processed by the filter if it is validated at the input of the filter. Figure 4.3 shows the structure of this pattern.

In the figure, the sensor is responsible for measuring data from the monitored entity. Once the data is available it is passed to the input processing unit, which transforms the measured data into some new form. This new data is then passed to the data transformation unit and at the same time to a data validation unit. If the data validation fails then two options are available, either the processing is stopped and the system waits for the next iteration assuming that the failure was transient, or the data validation unit attempts to fix the invalid data and passes this data to the data processing unit. The above steps are then repeated until the action is sent to the actuator. For the purpose of the system described by this thesis the validation unit simply stops the processing until valid data arrives, thus preventing incorrect actuation actions and leaving the system in an unmanaged state.

While not implemented for this thesis, another option for reliability in the autonomic system could be to use a redundancy pattern. Two types of redundancy patterns exist - homogeneous and heterogeneous. For homogeneous redundancy, multiple control loops are created all with the same structure. The control loops run either in parallel and use a voting procedure in order to reach a decision like in the Triple Modular Redundancy Pattern or run in a main/backup setup like in

the Switch To Backup Pattern [46]. An example of a homogeneous redundancy for the autonomic system would be creating two control loops with the same logic, and with a periodic iteration every 30 seconds. However, one would be started with a shift of 15 seconds from the other's start time. An actuation would be taken only if both control loops reach the same decision.

For heterogeneous redundancy, multiple control loops are created but with different structures. The downside of this design in normal real-time applications is that it requires an increased design and development time for the system. For the architecture presented here, as long as multiple options exist for the various components this is not a problem. The advantage of the heterogeneous redundancy pattern is that identical design and implementation errors are unlikely to appear in all the control loops. For an autonomic system architecture as described in this thesis, this could include for example building two control loops, one based on control theoretic approaches and one based on machine learning approaches. Like the homogeneous example, a decision would be taken only when both control loops agree.

The redundancy patterns are not implemented for this thesis, however due to the modular design of the architecture adding them is possible. The architecture would require an extra component which manages the voting process of the multiple control loops.

4.5.2 Fault tolerance

In real-time systems fault tolerance and reliability are strongly interlinked. For example, the Switch To Backup Pattern deals with a main/backup system where in the case of failure in the main control loop, the backup loop takes over. For the purpose of this thesis, they are separated since different mechanisms are used for each of them. Fault tolerance is achieved in the system by using error recovery methods. Each of the components periodically saves its state and internal structure and in the case of a complete failure of the component, of the application server it is running in, or even of the hardware the component is running on, the component can be restarted from the last saved checkpoint. This is achieved by adding a persistence capability to each component, where through some mechanism which is not specified in this section the capability can request to save its data. After a failure of the component, the persistent storage is checked for existing data and if

any such data exists the component loads the data and brings its internal state to that of the saved data.

While persisting its data the component also stores data regarding the location of other components it communicates with. Once it brings its internal state back to the one before the failure, the component communicates to the other components to notify them, that it is back online. In the period of time when the component is offline, the components communicating with it recognize the situation and stop transmitting data to it until they receive a notification that the component is back online and in the correct state. The persistent capability provides two interfaces - store and load. The store interface receives an XML description of what the component needs to save and the load interface returns an XML document. The persistent capability does not care what the XML document contains, it is the responsibility of each component to implement appropriate save/load functions which process the XML and map it to the components state and structure.

4.5.3 Synchronization

The third issue solved through the use of real-time software patterns is that of message synchronization between components. In order to ensure that the correct operation of the control loop takes place, the architecture infuses two real-time patterns for synchronization - one for synchronous messages and one for asynchronous messages. Even though all messages between components are sent via notifications, some communication instances require synchronous replies. For example, a coordinator that requires data from the model, must wait for the data to return before it can continue with the processing of the control logic.

4.5.3.1 Asynchronous communication

In the described architecture, asynchronous messaging between components is done via the message queue pattern. In the message queue pattern, messages between components are passed through the use of queues, which store the messages to be sent, or received until the next time the receiver or sender thread executes respectively. Data integrity for the messages is guaranteed by the use

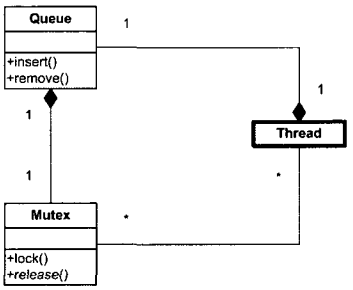

FIGURE 4.4: Message queue pattern

of a mutual exclusion semaphore (mutex) to guard against multiple, simultaneous access operations on the queue. The application of the above communication pattern is shown in 4.4.

In the above diagram, two or more threads which communicate through a message queue, would share one queue (or multiple ones - one queue for each type of message) in order to pass messages. The queue is owned by the receiving thread and messages received, via notification from other components, are stored in the queue. When the receiving thread becomes active, it retrieves data from the queue and processes it. In order to ensure that the data processed is valid a mutex is used to ensure that there is no synchronous access. If a mutex would not be used, both the sender and the receiver could access the queue at the same time. In such a case, if the receiver accesses the queue, while the sender is updating the queue with new information, the receiver would retrieve partial information that might not represent the correct information. In some cases, the receiver could interpret the missing data as zero or null, which would introduce further errors in the control loop. Even worse, such incomplete information could cause a crash or errors raised by the receiver, if it can not retrieve information that is required for further processing of data. In this example, by using a mutex if the receiver tries to read the data while the sender is adding new information to the queue, the receiver would be blocked until the sender finishes adding data to the queue, as the data in the queue would be incomplete. Because of the fact that the data is incomplete, the data can not be processed correctly. In this architecture, since the components can be distributed the sender and receiver are part of the same component. The sender represents the thread which receives notifications and the receiver is the thread which processes the information.

Furthermore, to ensure that the data is processed in the correct order the messages contain the timestamp of the sending component. Messages are added to the

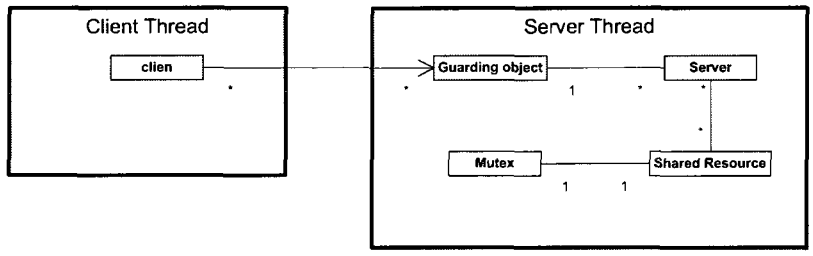

FIGURE 4.5: Guarded call pattern

queue based on the timestamp, with the oldest first. As the receiver processes the messages it stores the timestamp of the last processed message. Any new message with a lower timestamp than the last processed is discarded as old.

4.5.3.2 Synchronous communication

The guarded call pattern takes care of the cases where a synchronous message is required for the communication among components. Message queues can be used only for asynchronous messages as the data is added to the queue and taken from the queue, by each thread when the appropriate time arrives. This causes the data added to the queue to be processed by the receiving thread at its next execution. There are cases in an autonomic computing system where synchronous messages are needed. One example of such a case is the communication between a coordinator and a model. In a control theoretic control loop based on Kalman filters, the coordinator uses an iterative approach to calculate the estimated data and update the model. In each of the iterations, the coordinator estimates the data, updates the model, and verifies if the estimated values converge. Furthermore, the estimator uses the model also in order to do the estimates. The communication between components for this example has to be synchronous, as the coordinator has to wait for the estimated data returned from the estimator before it can continue its operation. The simplest solution is to make a synchronous invocation in the appropriate component. This however can lead to mutual exclusion problems, so the guarded call pattern also employs a mutex to prevent simultaneous access to the same resource. Figure 4.5 presents the application of the pattern.

In this diagram a client and a server communicate with each other. The client thread trying to access a resource in the server thread. The invocation is done across the server thread boundary, and is accomplished through the use of the guarding object. The guarding object has the role to combine all the method

invocations that use the same shared resource, in order for the resource to be protected across all possible method invocations. Like in the message queue pattern, the mutex is a blocking semaphore, which protects the shared resource across multiple invocations. The role of this pattern comes into play when more than one client attempt to use the same resource in a server thread. For example, an estimator and a coordinator which use the same model in order to do the processing. If the coordinator requests an update of the model at the same time as the estimator requests the values of the model, one of the two requests needs to be blocked until the other completes.

Unlike asynchronous messages, synchronous messages do not require timestamps, as a request can either be completed and a value returned or a request is blocked. In both cases until a reply returns to the initiator of the messaging, the initiator is blocked.

4.6 Common component infrastructure

All eight autonomic system components have a common infrastructure which manages the life cycle of the components. This infrastructure is outside the functional part of the components and is meant to allow clients to create, manage and destroy autonomic components. The infrastructure is composed of three parts, as shown in Figure 4.6.

Component Manager which is responsible for managing the components at run time by loading, removing or modifying the components. The *Component Manager* uses the management interface provided by the component in order to configure the component.

Component Loader which is responsible for the actual loading of a component, in response to a request from the *Component Manager*.

Component Repository which is responsible for storing the loaded component instances. A request for a component goes through the repository, which attempts to locate the component, and returns a way to access the component.

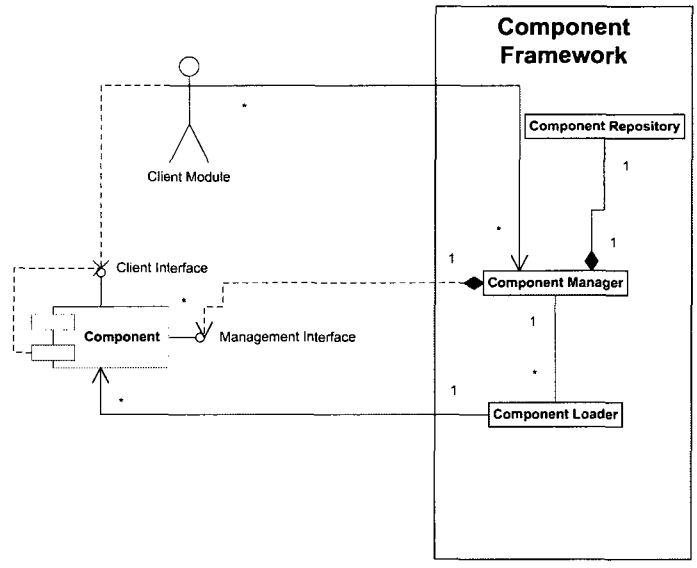

FIGURE 4.6: Common component management infrastructure

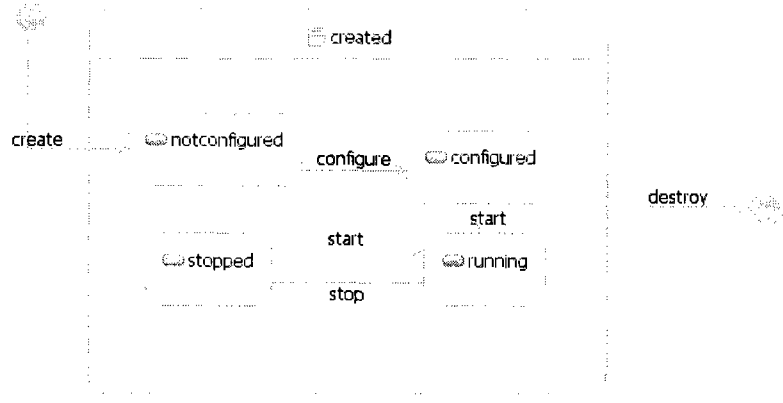

FIGURE 4.7: Component state chart

The first step in the operation of the infrastructure is for a client to request, either based on a human operator or an autonomic decision, the creation of a new component. The creation request goes to the *Component Manager* which finds the appropriate component interface in the *Component Repository* and uses the *Component Loader* in order to create a new instance based on the interface. The *Component Manager* uses the component's management interface in order to configure the component. Once configured, the new component is added in the *Component Repository* and the information about how to access the component is sent back to the client. The client can later request the start of the component. The start request goes to the *Component Manager*. The *Component Manager* finds the

appropriate component in the *Component Repository* and passes the request to the component. Once started, the status is returned to the client. Future functional requests to the component go to the component's client interface. However in order to access the component the client communicates with the *Component Manager* again. Like before the *Component Manager* passes the requests to the component. Finally when the component is no longer needed, a destroy request is sent to the *Component Manager*. The *Component Manager* uses the management interface to shutdown the component and then destroys it and removes it from the *Component Repository*. Figure 4.8 shows the sequence of events for component management, and figure 4.7 shows the state chart of components.

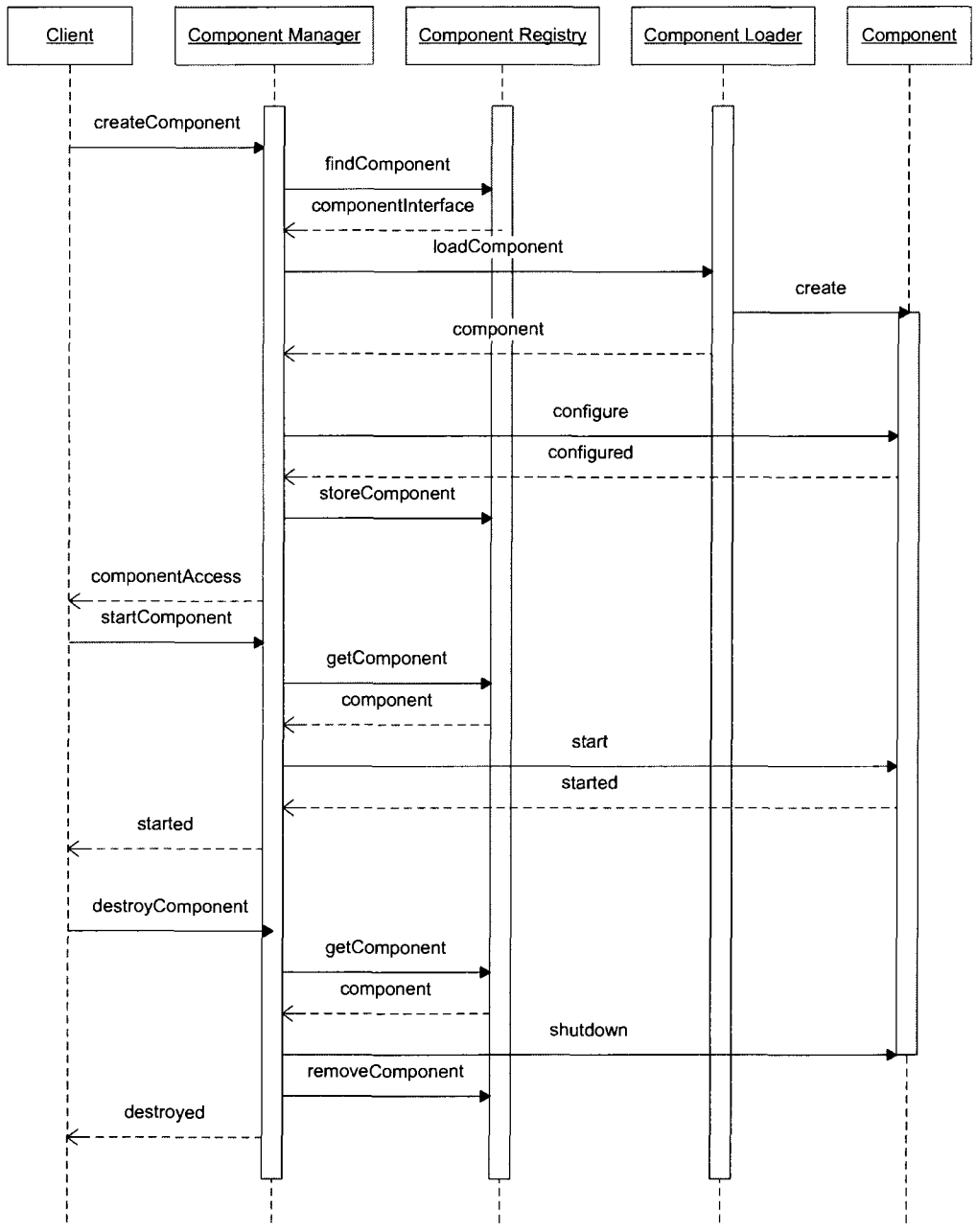

FIGURE 4.8: Component management sequence

Chapter 5

Low Level Design

The second step in using the MDD paradigm is to build a Platform Specific Model (PSM). The PSM defines the architecture based on the language of development, system on which to execute and methods of communication between components. As such the PSM represents a low-level view of how the system components are created and how various components communicate with each other. For the architecture presented in this thesis the PSM defines how each of the components is built, how the real-time patterns are implemented and how open standards are used.

The architecture developed in this thesis is based on the Java language and the components run in a Java Enterprise Edition (JEE) container. All communications between components are done via SOAP and web services. All components are created as WSDM resources and part of the WSDM specification is used to provide mechanisms needed by the system. Since WSDM is just a specification, an actual implementation of the standard is used in the form of Apache Muse [47], an open source project under Apache. The reason to choose WSDM as the deployment container is two folded. First of all, one of the key requirements for an autonomic system according to the IBM Manifesto is that an autonomic system must "function in a heterogeneous and open way, and implement open standards". The purpose of this requirement is that the management solution must be independent from the vendor which provides it and the systems that it manages, such that the solution can be applied to a variety of hardware and software systems from various vendors. From the point of view of the autonomic system a server should be seen as a server, no matter if it is a JBoss Application Server [48] or

a WebSphere Application Server [49]. The interfaces that provide data regarding the state of the server and its performance should be the same across a product line. Similarly, the management interfaces should be the same across a product line such that, for example increasing the thread pool is done the same way for all application servers. This is provided by the WSDM standard, since the WSDM standard provides "interoperable, base manageability for monitoring and control managers using Web services" [50]. The WSDM specification also provides a way to compose management resources from various capabilities, which is an utility that was used in the architecture, and makes use of other Web Service standards including Web Service Addressing (WS-Addressing) and Web Service Notification (WS-Notification).

5.1 Component Implementation

As stated, the component framework is developed in Java using Apache Muse. All the components have a common part, which is responsible for the management of the components and a separate part which defines the execution of the component. One component container can manage multiple component instances which run separately, each in their own thread. The WSDM standard specifies two levels of abstractions - the resource and the capability. A resource is a composition of capabilities, providing a unique cohesive web service endpoint where messages can be received. A capability is a collection of properties, operations, events and metadata that describe a certain functionality of the resource. Some capabilities are already provided by the Apache Muse framework, and some are developed for the autonomic system. Figure 5.1 taken from [47] shows the architecture of the Apache Muse framework. The isolation layer provides isolation from the web service container used by Muse. The router is capable of finding the resource that receives a request, and routes the request to that resource. Finally the resource routes the request to the correct capability which is responsible for that functionality.

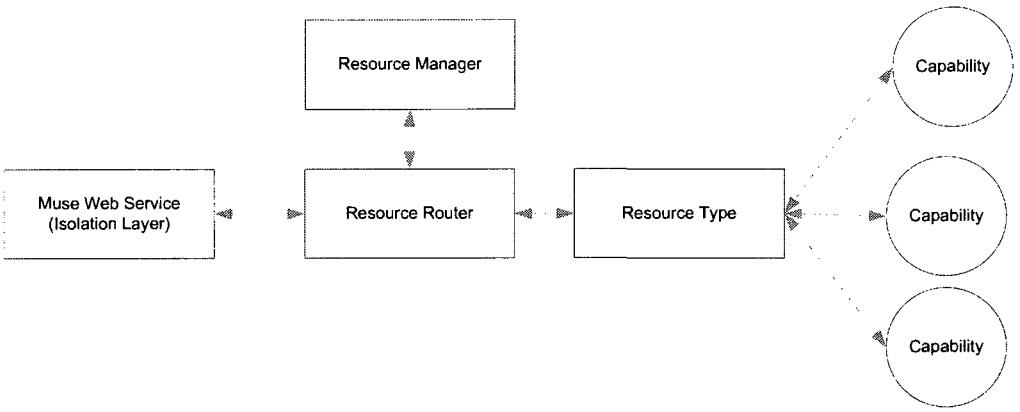

FIGURE 5.1: Apache Muse Architecture

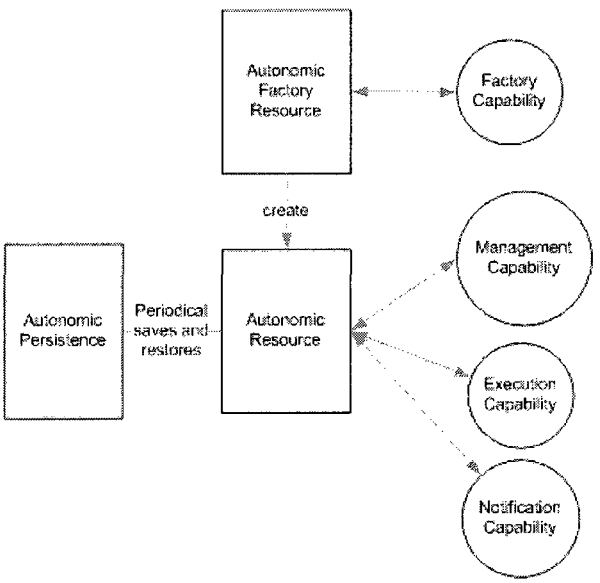

FIGURE 5.2: WSDM Based Component Architecture

5.1.1 Component Management Framework

The component management framework represents the common part of the eight autonomic system components. It implements the *common component infrastructure* described in the previous chapter. Since WSDM is used to develop the framework, each component will be composed by a group of WSDM resources. Figure 5.2 shows the structure of the component architecture implemented using WSDM. The component is split in two resources and one helper class:

Autonomic Factory Resource Singleton factory resource, responsible for creating instances of the components.

Autonomic Resource Resource which provides the functionality for one component instance.

Autonomic Persistence Helper class which provides the persistence mechanisms used for fault tolerance as described in Chapter 4.

The *Autonomic Factory Resource* has only one capability, the *Factory Capability.* This capability receives a request to create a new component based on a certain component type, with the component type representing the name of the class to create. The capability then uses Java Reflections in order to create an object based on the requested class and also creates and initializes an *Autonomic Resource* which encapsulates the given object. Once created a unique endpoint is created for the resource, and this endpoint is registered with the *ResourceManager* of the Apache Muse instance the factory is running under. The client which requested the creation receives back the unique endpoint of the created resource.

The *Factory Capability* has an extra responsibility, which takes place on its initialization. The initialization of the factory capability, which is based on the encapsulating resource's initialization , takes place when the container (i.e. server) in which the component resides starts. This extra responsibility is to discover the components that can be created by the factory and to register the components with the *Autonomic Registry.* The discovery is done by using class loader mechanisms in the JVM. Since all the components will implement the same interface, for example filters will implement the filter interface, this mechanism searches, using the class loader's paths, classes which implement the component interface. Once found, the class types - based on the class name are stored with the registry. When the factory shuts down, it asks the registry to remove all it's component types from the registry since creation of new components will become unavailable.

The *Autonomic Resource* is composed of three capabilities - management, execution and notification. The *Management Capability* provides management mechanisms for the resource including configuration, start, stop and remove interfaces. The configure function, configures a component that was previously created and changes the state of a component from *notconfigured* to *configured.* This operation is component specific as the configuration of a sensor is different from that of a filter, for example. The start operation moves the component from a *configured* or *stopped* state into a *running* state. While in the *running* state, the component receives messages and processes these messages as needed. The stop operation

moves the component into a *stopped* state, in which the component can not process messages. Finally the destroy operation removes the component from use; only a previously stopped component can be destroyed.

The *Management Capability* provides three other helper operations. The createNotificationProxy operation creates the notification mechanisms used by the component, which will be explained later. The persist and reload operations are used in order to communicate with the persistence layer. The persist function is periodically called by the persistence layer in order to save the state of the component, and the reload function is called by the persistence layer on startup if there exists saved data for the component.

The *Execution Capability* provides a wrapper to the functionality of the component and simply passes messages coming to the component from the outside to the class implementing the functionality.

The *Notification Capability* provides the notification mechanisms between components. The notification mechanism is based on a producer/subscriber model. Each notification producer has a number of topics that it offers; subscribers can then choose to subscribe to some or all the topics provided. When data is available for a topic, the producer sends notifications containing the data to all the components that are subscribed to the topic. The *Notification Capability* is built on top of the Apache Muse framework through the use of a NotificationProducer on the sender side and a NotificationClient on the receiver side. The producer and client provide partially the mechanisms used for notifications between peer resources. These mechanisms are extended on the receiver side in order to provide the full functionality required by the autonomic framework. As such, the receiver side of the notification is composed of three interfaces and two implementation classes that are responsible for creating the notification client, starting/stopping the client as needed, and processing the information before it is passed to the resource's appropriate capability. The interfaces and classes are:

NotificationEnabledResource Any autonomic component that wishes to accept notifications must implement this interface. The interface adds one function - setNotificationProxy, which adds a reference to the proxy used to set up the notification mechanisms.

NotificationProxy and MuseNotificationProxy The NotificationProxy interface and its MuseNotificationProxy implementation encapsulate the logic

required to set up the notification mechanisms. The interface specifies the high-level communication with the other components, while the MuseNotificationProxy class implements the logic specific for the Apache Muse notification mechanisms. The reason to separate the interface from the Muse implementation is in order to allow the components to use other notification mechanisms without changing the component logic. The interface defines six functions - setDataRetriever, which attaches a retriever to the proxy; setProducerInformation - which adds the information regarding the endpoint of the producer that will receive the subscription request; setTopics - which sets the information regarding what topics the client wishes to accept; start/stop - which start and stop the proxy; and getUniqueID - which returns the unique identification of the endpoint receiving the notifications.

NotificationBasedDataRetriever must be implemented to specify the processing logic for any incoming notifications. The interface specifies two functions that must be implemented - setData, which is called by the proxy to pass a new received notification and getUniqueID - which returns a unique identification of the producer that is sending the notifications. The *NotificationBasedDataRetriever* also implements the message queue mechanism described in Chapter 4.

MuseListener is an implementation of the *NotificationMessageListener* interface in the Muse framework. It implements two methods - accepts which receives a message and decides if it accepts it based on the topic in the message; and process which processes the content of an accepted message and adds the data in the retriever.

In this structure a *NotificationEnabledResource* can have any number of *NotificationProxies*, and a *NotificationProxy* is attached to one *NotificationBasedDataRetriever*. Figure 5.3 shows the structure of the notification mechanisms and figure 5.4 shows the sequence diagram. The ComponentImplementation implements both the NotificationEnabledResource interface and its own ComponentInterface, which specifies the functions required to be implemented by the component, like filtering for a filter.

When a component is first configured, it creates the notification mechanisms by creating the listener, the retriever and the proxy. Once the start command is received, the component starts its notification proxy which in turn starts all the

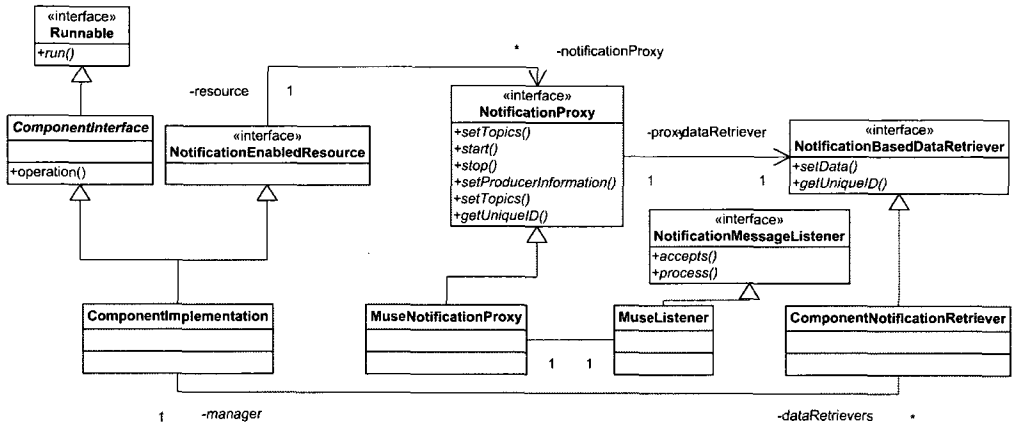

FIGURE 5.3: Notification Mechanisms

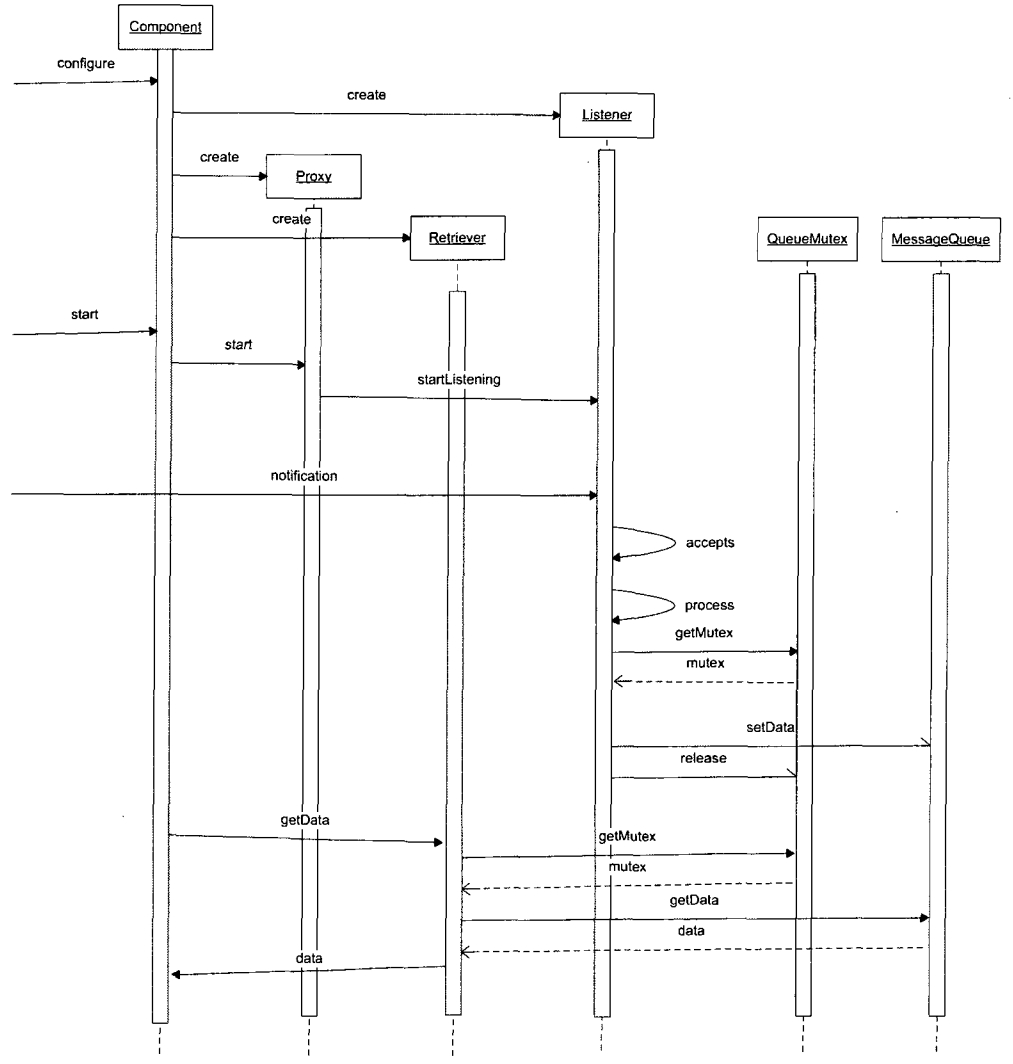

FIGURE 5.4: Notification Mechanisms Sequence

listeners it has attached. When a notification arrives later, the Muse framework sends the notification message to the listener, which decides based on the topic if it should accept it or not. If it accepts it, the listener processes the message and attempts to access the MessageQueue by trying to get the mutex. If it can not obtain the mutex the execution blocks until the mutex becomes available. If it can get the mutex, it simply adds the data to the queue and releases the mutex. When the component requires the data later, it also tries to obtain the mutex, and once it has obtained the mutex it retrieves the data and releases the mutex. Once a stop request is received by the component, it passes the request to the proxy, which also halts the listening operation.

The *Autonomic Persistence* helper classes provide the mechanisms to periodically save information from the autonomic components. This mechanism is added by extending the default Apache Muse persistence logic. The functionality is provided by two classes that act together to periodically save the information about a persistable autonomic component and restore the state of such a component in case of failure. The two classes are:

AutonomicComponentFilePersistence which extends the default Muse framework file persistence logic in order to act as a persister/restorer for an autonomic component.

PeriodicPersister which is a thread that runs periodically and saves the information about the required component.

Furthermore, the persistence functionality requires that each component that must be persisted implement two functions: *persist* and *reload*. The *persist* function returns a valid XML document that can be saved in a file, and the *reload* function accepts an XML document and knows how to recreate the autonomic component from it. In order to separate the persisting logic from the components, the call of these methods is done via the Java Reflections API. Components that require persistence only need to implement the two methods and the system knows how to persist them; components that do not require persistence do not need to implement the two methods and the lack of the two methods will have no impact on the system. Figure 5.5 shows the structure of the persistence mechanisms. The AutonomicComponentFilePersistence extends the Apache Muse class AbstractFilePersistence, and creates/destroys the PeriodicPersister class, which implements the Java Runnable class and provides the periodic threading logic.

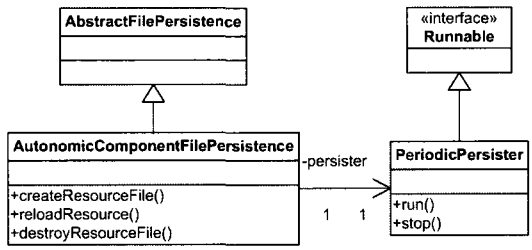

FIGURE 5.5: Persistence Mechanisms

The *component management framework* provides one extra functionality which is not part of the *common component infrastructure* - the validation framework. The validation framework acts as an implementation of the *Reliability* pattern described in Chapter 4. The goal of the validation framework is to allow developers to implement their own logic required for the validation of messages as well as the actions to take when validation errors occur. The implementation is composed of the following classes and interfaces:

MessageValidationSubsystem which is the main class used to drive the message validation. The subsystem uses a filter chain pattern in order to pass the validation through a number of validation filters. The class receives as input the message to be validated, and it provides as output the validated message, which can be the initial message if there are no problems, a repaired message if it is possible to repair a damaged message, or an empty message if the validation fails and no repair is possible.

ValidationFilter The validation logic is composed of a number of validation filters, such that the validation can be chained, if needed. Each validation filter implements one type of validation logic. For example, one filter would ensure that all the required data is in the message, and another would validate if the data are within the expected thresholds. The validation filter uses the two following interfaces to perform it's functionality.

MessageValidator This interface is the interface to be implemented by the validators. Its role is to validate the message, or part of a message and return the part that is invalid if such a part exists. A filter uses a class implementing this interface in order to find errors in the messages. If errors are found the class implementing the next interface is called.

ValidationErrorAction This interface is the interface to be implemented by error recovery mechanisms, in order to specify the action to take in case a

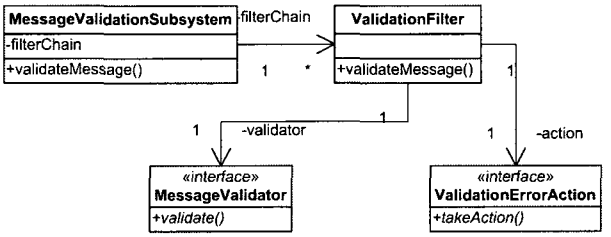

FIGURE 5.6: Message Validation Mechanisms

validation error is found. The message validator can have a list of possible actions to be taken depending on the error in the message. Based on the action it decides to use, it either fixes the message, or notifies the validator to discard the complete message.

Figure 5.6 shows the structure of the validation mechanism and figure 5.7 shows the sequence of events in a system with two validation filters. The mechanism is applied at both the input and the output of all the components.

The processing of the validation framework starts when the subsystem receives a message to be validated. The message is passed to the first validation filter in the list, which determines if the message is valid or not by using its message validator. If the message is valid an empty message is returned to the subsystem. The subsystem then passes the initial message to the next filter. This filter again uses its message validator to determine if the message is good or bad. If the message is not good, the validator returns the part of the message that is invalid. The filter then uses its validation error action, to which it passes the invalid part of the message in order to fix it. The error action examines the incorrect part and determines if the error can be fixed. If it can fix it, it returns the fixed message and the filter changes the invalid part with the correct part. If it can not fix it, the error action notifies the filter to discard the message, and the filter notifies the validation subsystem to discard the message.

A number of extra capabilities are provided directly by the Muse framework.

1. ResourceTermination - allows the destruction of a resource either immediately or somewhere in the future

2. ResourceProperties - allows the querying of what properties exist, and also the setting and retrieving of resource properties

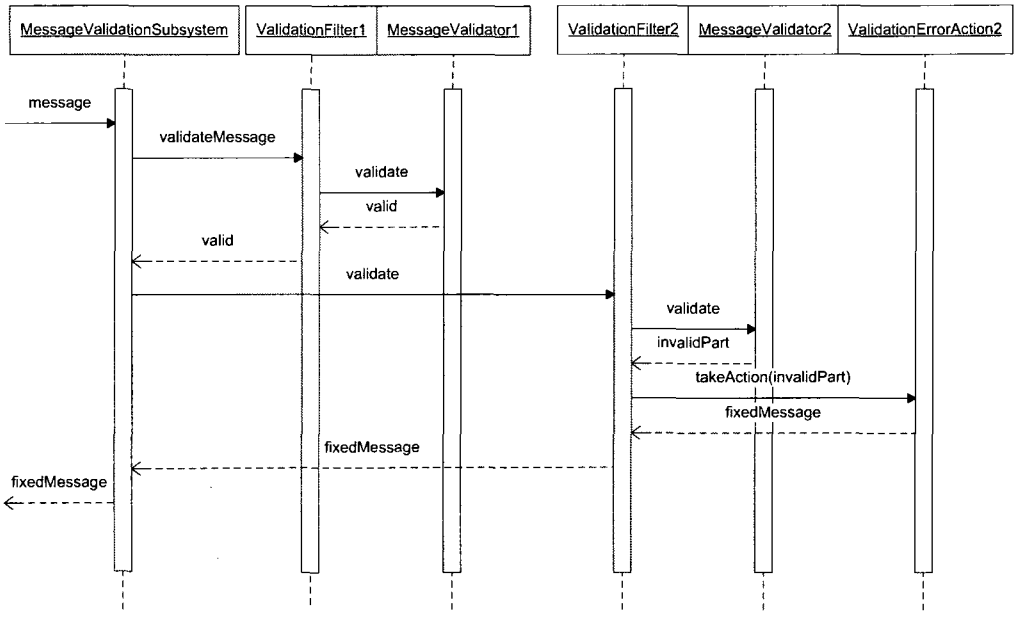

FIGURE 5.7: Message Validation Mechanism Sequence

3. Metadata - specifies mechanisms used to retrieve metadata about a deployed resource

4. State - specifies the state of the component, and allows the retrieval of the state by clients

5. ResourceId - generates a unique resource ID for each instance of a resource

6. Addressing - provides unique addressing for each resource that resides at a certain endpoint, and ensures that messages reach the correct resource

5.1.2 Sensor component

The sensor component has one goal, to retrieve data from the monitored resource and process the data to be passed forward to the filter. The data must be in its original raw form, which means that any processing must only include taking the data and transforming it into the data structure used to pass messages between components - WSDM format in this case. The processing must not include any kind of changes to the data values retrieved from the underlying monitored system, as that is the responsibility of the filter. The sensor can however filter the information that it can send such that a client of the sensor only receives the information that it requires. For example, a sensor for an application server which

uses JMX to retrieve the data can provide any information available through JMX to the filter. Such a sensor can however be configured to filter the data that it sends to the autonomic computing filter in order to minimize the data sent and processed. In the case of an autonomic loop that is interested only in the metrics of a certain web application, the sensor can filter the JMX data and send only the data about that web application. The measurement filters are set up when the sensor is configured.

At the same time, one sensor component can be composed of multiple probes, where each probe monitors and retrieves data for one specific part of the system. In the case of an application server, there could be one probe monitoring the response time, number of clients, throughput, etc. of a certain application, and another probe monitoring the connection to the backing database. As such, the sensor component is split into the following classes and interfaces:

SensorFactoryCapability which is responsible for creating the sensor resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. The capability receives a list of MeasurableData objects specifying what the rest of the control loop is interested in being monitored, as well as a numeric value specifying how fast the data should be gathered. The factory also provides to clients a way to query what types of data it can monitor.

SensorCapability which is responsible for configuring the sensor, and also starting and stopping the sensor. This implements the *Autonomic Resource* mechanisms. Configuring the sensor includes creating the appropriate probes and filters, setting the synchronization mechanism for the probes, as well as setting the notification producer endpoint which accepts subscriptions from clients.

SensorProbe which represents one probe that is used to gather data from one part of the system.

SensorFilter which is responsible for filtering unnecessary data retrieved by the probe. In the cases where one probe could retrieve more data than what is needed, because the underlying monitoring mechanism it uses returns too much data, the filter will remove any extraneous data.

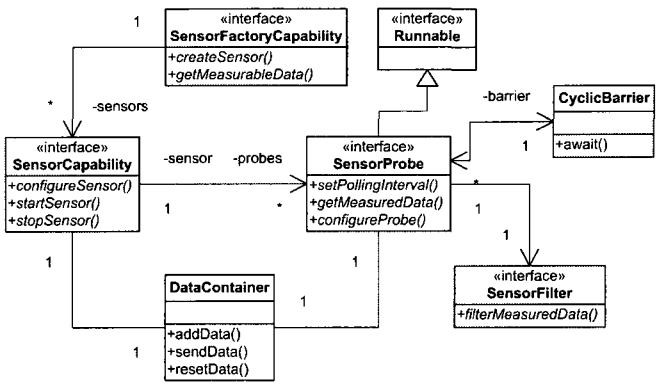

FIGURE 5.8: Sensor Structure

SensorDataContainer which creates a buffer at the sensor in order to store the data from multiple probes, such that when all the probes have returned their available data, it can all be sent together to the filter.

Due to the composition of the sensor from multiple probes, the probes must be synchronized such that the data returned by the probes represents almost the same moment in time. If probes were allowed to execute each in its own thread without synchronization, in time the executions would drift away from each other. In order to achieve this a cyclic barrier is used to ensure the synchronization. The cyclic barrier allows a number of threads to synchronize with each other, by setting a common wait point. Once all the probes reach the barrier, they are released in order to gather the information. Figure 5.8 shows the structure of the sensor component.

The sensor component contains one *Factory* which can create multiple sensors. Each sensor can have one or more probes that do the actual measuring. The sensor and the probes share a *DataContainer* in which the probes add the data and from where the sensor takes the data. Each of the probes also has a filter in order to remove extraneous data from the data retrieved from the underlying resource. All the probes also share a *CyclicBarrier* used in order to synchronize their execution.

Figure 5.9 shows the sequence of events required to create a sensor, start it, and the internal sequence of the sensor for one iteration. In this example the sensor has two probes, a local probe and a remote probe. In the first step, an autonomic system manager would request from the sensor factory which instances can be measured. Once a reply is received the manager can select the metrics to measure, and request

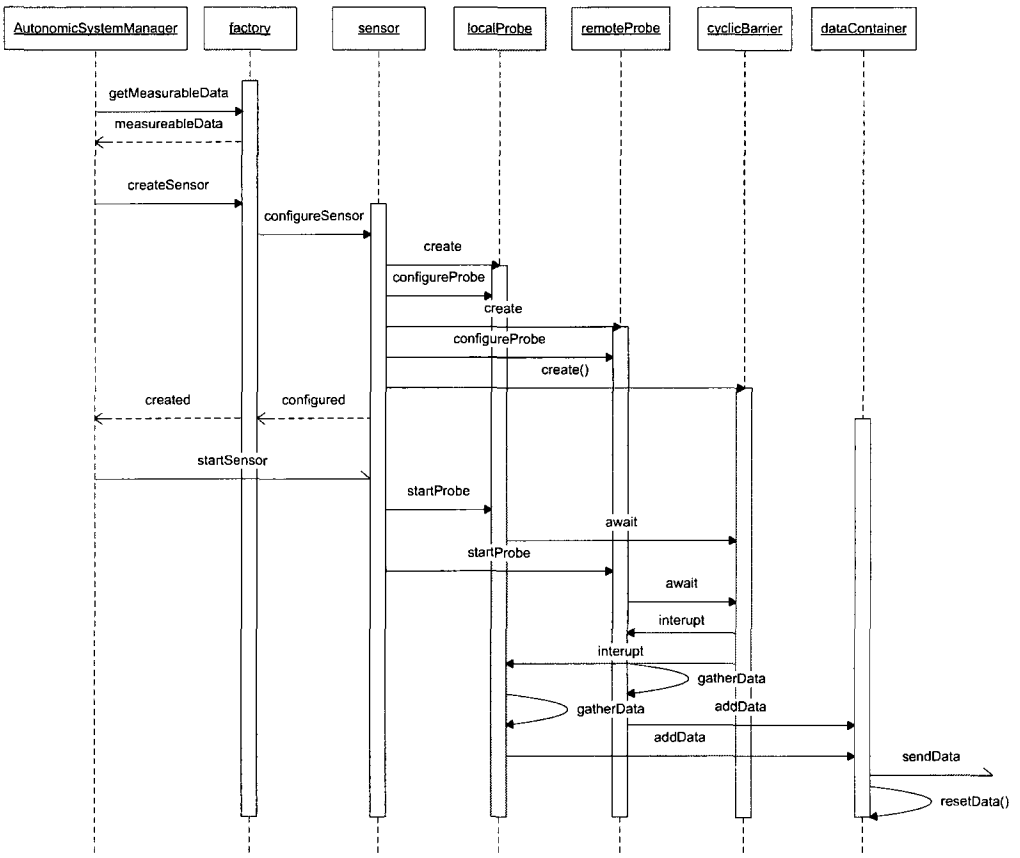

FIGURE 5.9: Sensor Sequence

the creation of a sensor. Upon reception of the create request, the factory creates a new sensor and passes the information about what is to be measured to the sensor via the configure request. In order to configure itself, the sensor analyzes what it needs to measure and creates the appropriate probes - in this case a local probe and a remote probe. After configuration, the sensor returns that it has configured itself. The manager later requests that the sensor starts, which results in the sensor starting its probes. The probes reach the wait point in turn, and once both have reached it, they are released to gather data. After the data is gathered and filtered, each of the probes adds its data to the common data container, which is synchronized to ensure that the probes do not overwrite each other's data. Once all the data is in, the data container sends the data to the autonomic filter (the data goes through the validation filter, if any) and the process repeats again after the sampling interval passes.

5.1.3 Filter component

Like the sensor component, the filter component has only one role - to transform the measured data into data that can be used by the rest of the control loop. In order to provide a certain degree of reusability and modularity, the autonomic filter implements a filter chain pattern, where the data is passed successively through a number of filters, each filter taking one action to modify the data. For example, in the WebSphere Application Server cluster example, the data coming from each of the sensors represents the total values for a certain metric since the server had started. What is required by the rest of the control loop is an incremental value for the previous pooling period. Furthermore, the control loop requires a number of metrics that are not directly available from the sensor, but which can be calculated from the measured metrics. The control loop also requires an average value for the whole cluster and not distinct values per server. Each of the above calculations can be abstractized into three separate filters, which can then be chained. The autonomic filter component is split into the following interfaces and classes:

FilterFactoryCapability which is responsible for creating the filter resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. In order to create a filter resource the capability receives a list of the filters to use in the chain. The factory also provides clients a way to query what types of filters are available.

FilterCapability which is responsible for configuring the filter, and also starting and stopping the filter. This implements the *Autonomic Resource* mechanisms. Configuring the filter includes creating the appropriate filter chain, setting the notification clients to the sensors, as well as setting the notification producer endpoint which accepts subscriptions from its own clients. Starting the filter resource is done by subscribing the notification clients to the appropriate sensors.

FilterManager which acts as the binding mechanism between the data coming from the sensors, the filter chain and the push of the filtered data to the coordinator. It ensures that the filter has received the data from all the sensors before it begins filtering. Also allows for the runtime addition and removal of sensors that must send their data for filtering, and takes care of creating the appropriate notification client mechanisms for each of the sensors.

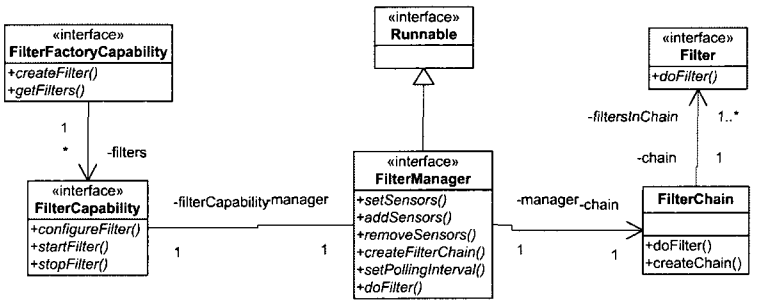

FIGURE 5.10: Filter Structure

FilterChain which defines a certain chain of filters and ensures that the data is passed successively through the filters. The filter chain also stores previously used data from the sensors in case the data is later needed. The data is stored based on a predefined buffer size.

Filter which is the base of a chain and provides the main logic for one filtering operation. In the example above, the three filters would be a difference filter, to calculate the incremental value; an averaging filter, to average the data across all the servers in the cluster; and a computational filter, to calculate the values that are not available from the servers.

Figure 5.10 displays the structure of the filter interfaces.

The filter component contains one *Factory* which can create multiple filter resources. Each filter has one filter manager, which is a Runnable instance and periodically drives the execution of the filter chain. Each manager has an attached *filter chain* describing the order of execution of the filters. Finally, each *filter* can perform one action - doFilter, which takes measured data or filtered data as input and outputs filtered data.

Figure 5.11 shows the sequence of events required to create a filter, start it, and the internal sequence of the filter for one iteration. The filter in this example has two external sensors and two filters in the chain. In the first step, an autonomic system manager would request from the filter factory what filters can be created. If the required filters can be created, the manager asks the factory to create a filter chain based on a certain filter sequence. The factory creates a filter resource, which creates a filter manager, and asks the manager to configure itself based on the filter sequence. The manager creates a chain and configures the chain with the given sequence, however it is the filter chain's responsibility to create

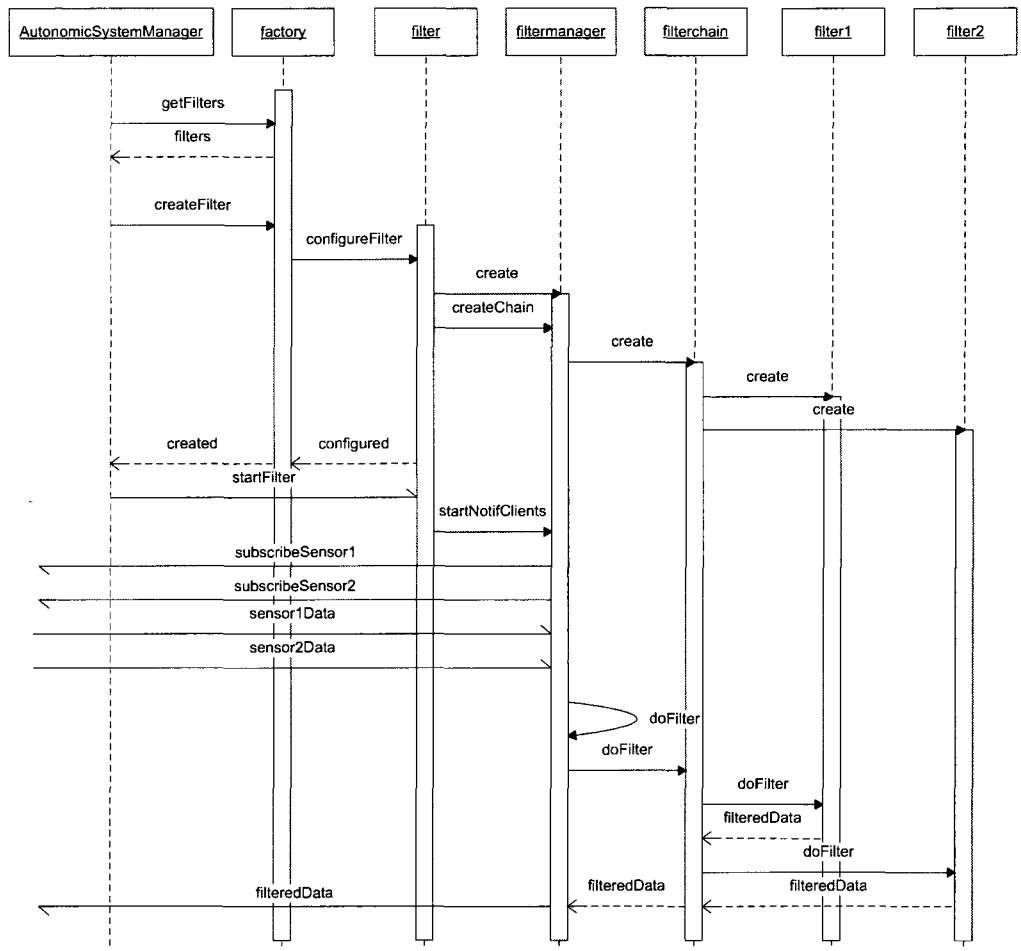

FIGURE 5.11: Filter Sequence

the filters. Once everything has been created, the filter resource returns to the autonomic manager that it is configured and ready to start execution. Once the start command arrives, the manager subscribes to the sensors whose data it must process. At a later time, the sensors send their data to the filter manager. Upon reception of all data, the filter manager starts the doFilter action, which is passes to the filter chain. The filter chain, in turn asks each filter to process the data based on the previous filter output. Once the last filter returns the filtered data, the filter manager uses its notification producer in order to send the data to its subscribers.

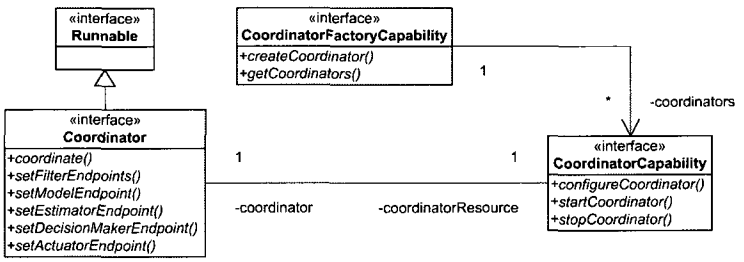

FIGURE 5.12: Coordinator Structure

5.1.4 Coordinator component

The coordinator component acts as the central brain of the control loop, and contains knowledge about the way in which to drive the other autonomic components to achieve the goals. Because of this, the structure of the coordinator is very simple, as different systems will have varying requirements. What the coordinator provides are the communication links with the other components. How those links are used, the order in which they are used and even if they are used is up to the developer of the control loop. For example, a simple self-protecting control loop that detects unauthorized access would not require an estimation of the future state, and could reach a decision based directly on the filtered data. A more complex system that self-optimizes would require an estimator to determine the future state of the system, such that it can decide what action to take. However, even the solutions for such problems are varied. A machine learning self-optimization control loop would get the new data, use the model and estimator in order to classify the data, possibly update the model with the new data by learning from it and pass the classification to the decision maker. An iterative control loop based on control theoretic methods would take a different approach. After receiving the data, it would successively estimate and update the model until a good enough estimate is found. Once this estimate is found, the coordinator would then request a decision.

Since all these logical flows are different, the coordinator does not impose any required order of events, other than a filtered data input being received, resulting in a coordination action whose output is a decision that is passed to the actuator. What the coordinator offers is as mentioned, the links between the components, and a way to define the driving logic between the communications. The structure of the coordinator interface is shown in figure 5.12.

CoordinatorFactoryCapability which is responsible for creating the coordinator resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. In order to create a coordinator resource the capability receives a list containing the other components' endpoints. The factory also provides clients a way to query what types of coordinator logic is available to create.

CoordinatorCapability which is responsible for configuring the coordinator, and also starting and stopping the coordinator. This implements the *Autonomic Resource* mechanisms. Configuring the coordinator includes setting the notification clients to the other resources, as well as setting the notification producer endpoint which accepts subscriptions from its own clients. Starting the coordinator resource is done by subscribing the notification clients to the appropriate components.

Coordinator which acts as the brain of the system. Upon reception of data from the filters, the coordinator periodically executes its only method - coordinate. This method contains the logic of how the other components should communicate with each other.

Figure 5.13 shows the sequence of events required to create a coordinator, start it, and the internal sequence of the coordinator for one iteration. In the first step, an autonomic system manager would request from the coordinator factory what coordinator resources can be created. If the required coordinator can be created, the manager asks the factory to create a coordinator of the given type. The factory creates a coordinator resource. At a later time, the coordinator resource through its management capability receives a configuration request, which contains all the endpoints it must communicate with. Since the other components must be created in order for the coordinator to configure itself, the coordinator is the last configured resource. Some of these endpoints can be null and the system will run fine without them, if the coordinator logic does not use them. Once configured, when a start message is received, the coordinator resource subscribes to its clients and starts its Runnable object. If all the data from the filters is available, the *coordinator* instance calls coordinate on itself, which executes the logic and once the execution is completed, it sends a decision to actuator if a decision has been reached. If no change decision is reached, the coordinator goes back to waiting for filtered data.

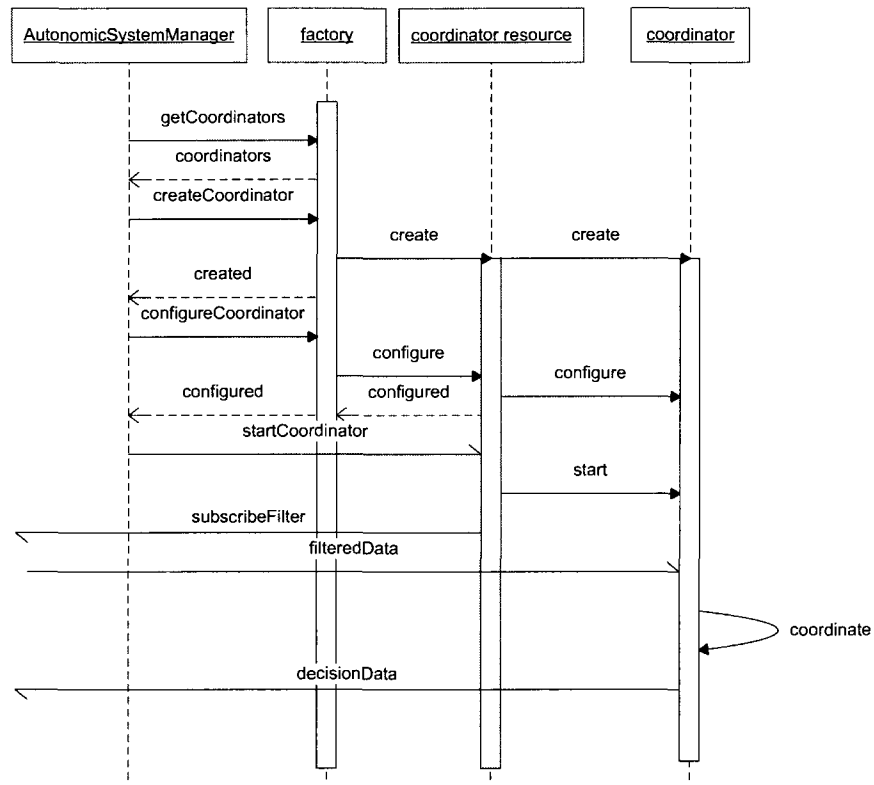

FIGURE 5.13: Coordinator Sequence

5.1.5 Model component

Similar to the coordinator, the model component does not enforce how the model of the underlying system is stored or represented. Like the coordinator, the models that can be used in autonomic systems are varied. The structure of a neural network model is completely different from that of a queuing model or from that of a Bayesian model. Furthermore, the action that can be taken on each of the models are different. Since some actions might not make sense for some models and it is the responsibility of the coordinator to know how to interface with the model, not all of the functions defined by the model interface have to be implemented.

In the case of the example control theoretic model, the model is represented by a set of mathematical equations, which map the measured inputs impact on the controlled outputs. The model is also a dynamic model, in that it is updated based on the input values and output values, such that its divergence from the actual system is minimal. Figure 5.14 shows the model's structure. The model allows the setting of model variables from outside, for example based on measured data or

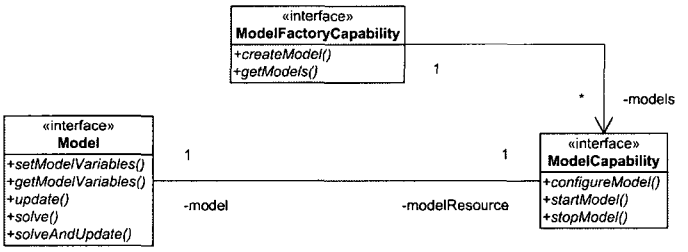

FIGURE 5.14: Model Structure

estimated data. The model also allows the updating of the model. The difference between setting the model variables and updating the model, is represented by what values are being changed. A model is composed of two parts: the parameters of the model and the variables in the model. In the case of the Kalman filter model, the parameters are the Jacobian matrices [25], and the variables are the state matrices. Updating the model modifies the Jacobian matrices while setting the variables sets the state matrix. At the same time, if it makes sense for the specific model being used, the model allows requests to solve the model, or solve and update the model after the solution is found. Solving the model, means for a given set of input values, finding the output values. Solve and update is different in that once solved, the model is updated with the given input/output values.

ModelFactoryCapability which is responsible for creating the model resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. In order to create a model resource the capability receives only the type of model to create. The factory also provides clients a way to query what types of models are available to create.

ModelCapability which is responsible for configuring the model, and also starting and stopping the model. This implements the *Autonomic Resource* mechanisms. Configuring the model includes setting the notification client to the coordinator, as well as setting the notification producer endpoint which accepts subscriptions from its own client - the coordinator. The model also receives at configuration the initial values of the model's parameters and variables. Some models will also require their internal structure to be defined and this is done via the configuration as well. For example, for a neural model one can define the number of inputs, hidden layer and outputs. Starting the model resource is done by subscribing the notification clients to the appropriate components.

Model which stores the model data and structure and allows the appropriate operations on the model. The component has setter and getter for its model variables, as well as solve, update and solveAndUpdate operations.

Figure 5.15 shows the sequence of events required to create a model, start it, and the internal sequence of the model for a certain request pattern from a coordinator. In the first step, an autonomic system manager would request from the model factory what model resources can be created. If the required model can be created, the manager asks the factory to create a model of the given type. The factory creates a model resource and returns its endpoint. At a later time, the model resource through its management capability receives a configuration request, which contains the configuration of the model and the initial variables of the model. The model is configured first, and once its structure is created, the initial values are set. Once a start message is received, the model subscribes itself to the coordinator in order to receive messages. At a later time in the iteration, the model receives first an update of the model variables via its setter, followed by a request to solve and update the model. Once finished the model returns the modeled values to the coordinator.

5.1.6 Estimator component

The estimator is, like the coordinator and the model, a component that depends on what type of control logic is implemented. Its goal is to try and determine what the future state of the system will be, based on the measured variables and the model state. The estimation function could be a classification of the data points for a machine learning algorithm or a mathematical function of what the values of the measured variables will be somewhere in the future for a queuing model or control theoretic control loop. The estimation is needed especially in the cases where the provisioning action is long, on the order of minutes and reactive decisions would result in a breach of the required performance. In the cases where an estimator is not necessary, the estimation step can be skipped by the respective coordinator. The estimator's structure is very simple - the interface requires only the implementation of an estimate function. Figure 5.16 shows the structure of the estimator component:

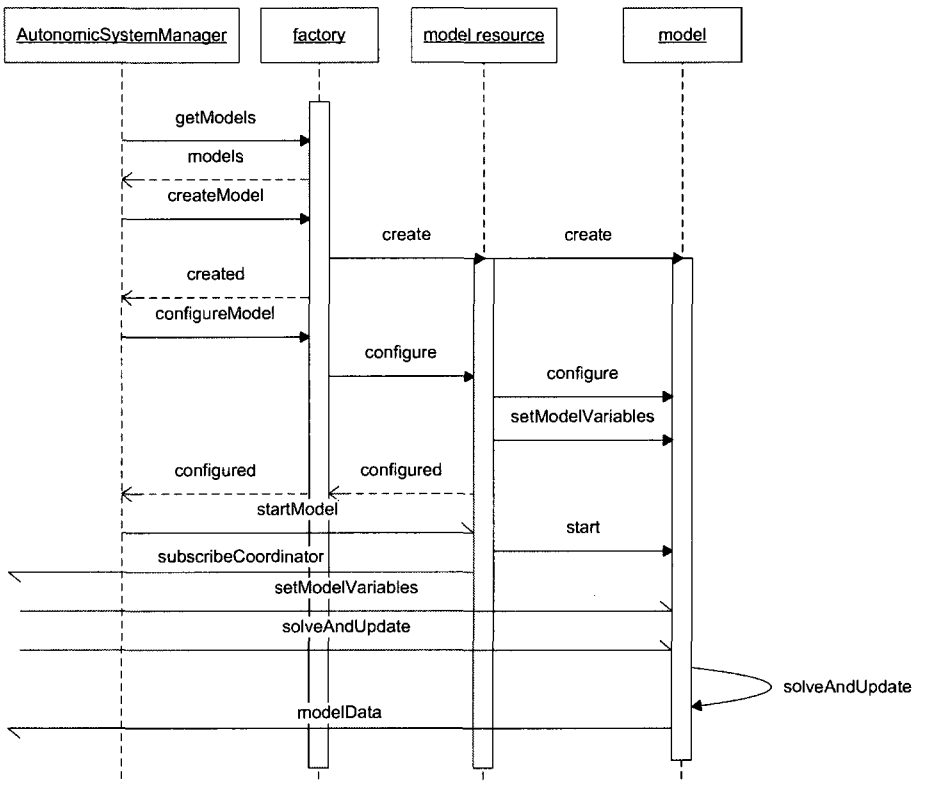

FIGURE 5.15: Model Sequence

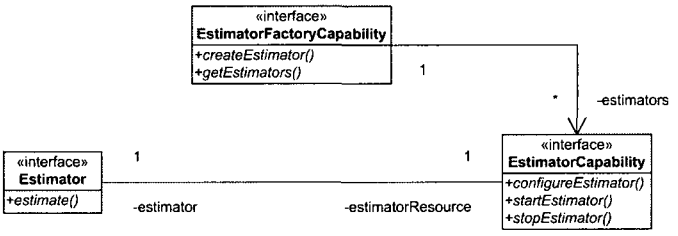

FIGURE 5.16: Estimator Structure

EstimatorFactoryCapability which is responsible for creating the estimator resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. In order to create an estimator resource the capability receives only the type of estimator to create. The factory also provides clients a way to query what types of estimators are available to create.

EstimatorCapability which is responsible for configuring the estimator, and also starting and stopping the estimator. This implements the *Autonomic Resource* mechanisms. Configuring the estimator includes setting the notification client to the coordinator and model, as well as setting the notification

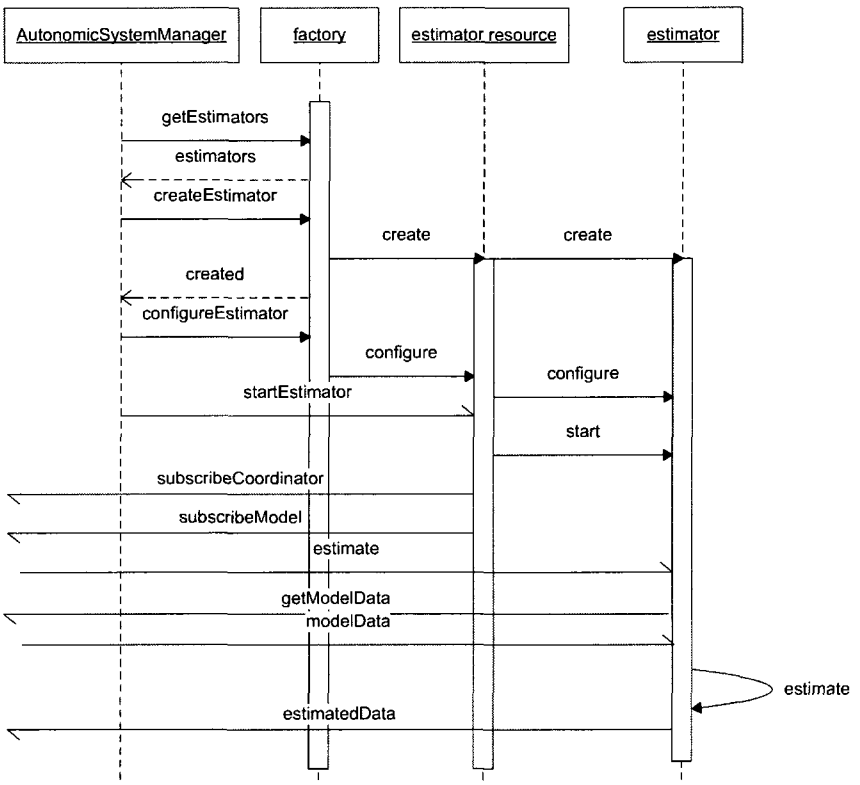

FIGURE 5.17: Estimator Sequence

producer endpoint which accepts subscriptions from its own client - the coordinator. The estimator receives information about the model's location because it communicates directly with the model to perform its estimates.

Estimator which computes the future state of the system. In one iteration it is possible for one estimator to receive multiple estimate requests from a coordinator.

Figure 5.17 shows the sequence of events required to create an estimator, start it, and the internal sequence of the estimator for one estimate request received from a coordinator. The creation/configuration/start operations are similar to the model, with the extra step of subscribing to a model. Once started the estimator receives an estimate request, and in turn requests the model state from the model. Once the model data is received, it estimates the future state and returns to the coordinator the future state of the system. For complex systems, the estimator's estimate method allows a coordinator to specify for how far in the future the estimate should be.

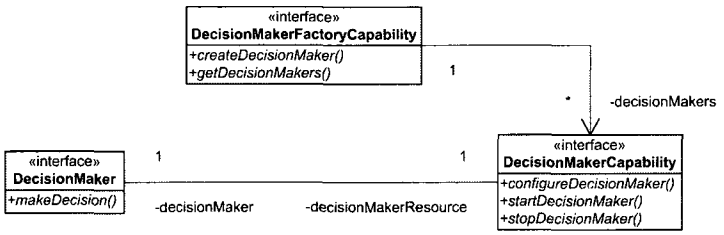

FIGURE 5.18: Decision Maker Structure

5.1.7 Decision Maker component

The decision maker is responsible for analyzing the state of the system and using its internal structure and that of the model to determine if any changes have to be made in the system. In the case that a change has to be made, the decision maker returns a change plan that is to be passed to the actuator for execution. The internal structure of the decision maker depends on the type of method used to build it, and as such the interface only requires one method - makeDecision, which receives a request from a coordinator and returns a Decision datastructure. Figure 5.18 shows the structure of the decision maker's capability.

DecisionMakerFactoryCapability which is responsible for creating the decision maker resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. In order to create a decision maker resource the capability receives only the type of decision maker to create. The factory also provides clients a way to query what types of decision makers are available to create.

DecisionMakerCapability which is responsible for configuring the decision maker, and also starting and stopping the decision maker. This implements the *Autonomic Resource* mechanisms. Configuring the decision maker includes setting the notification clients to the coordinator and model, as well as setting the notification producer endpoint which accepts subscriptions from its own client - the coordinator. At configuration time, it is also possible to set the thresholds necessary to reach a decision.

DecisionMaker which creates the decision action based on the model structure and the estimated data.

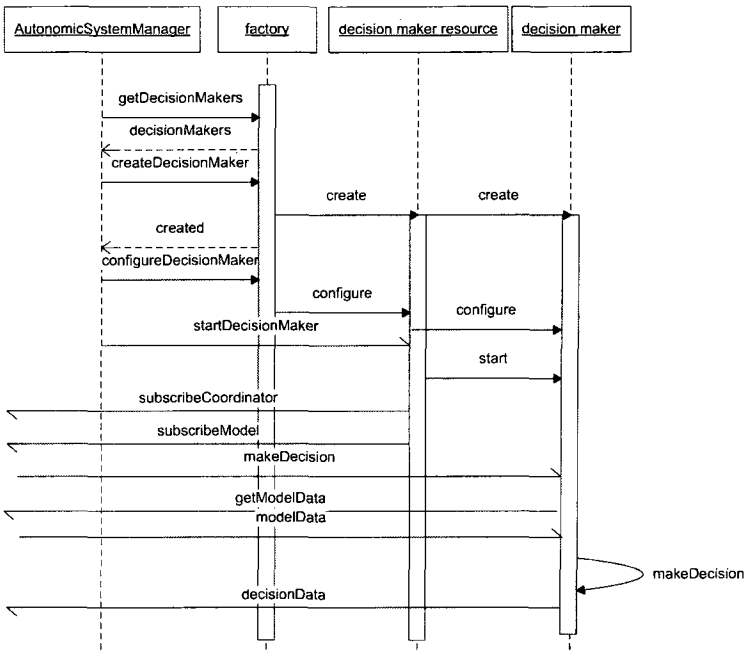

FIGURE 5.19: Decision Maker Sequence

Figure 5.19 shows the sequence of events required to create a decision maker, start it, and the internal sequence of the decision maker for one execution request received from a coordinator. The creation/configuration/start operations are similar to the model. Once started the decision maker receives a makeDecision request based on an estimation made by the estimator. The decision maker then uses the model data to reach a decision. If no changes are to be made, the decision maker does not send anything to the coordinator. If changes are to be made, the decision maker returns the decision plan.

5.1.8 Actuator component

The actuator is the component responsible for executing the actions required to bring the decision to fruition. It is also responsible for notifying the adaptor about what entities were modified, such that the adaptor can modify the control loop if needed. In the cases where an execution engine already exists, the actuator can provide a bridge between the autonomic control loop and the execution environment. For example, in a case where IBM's Tivoli Provisioning Monitor (TPM) is present in the environment, the actuator can provide a communication bridge to TPM and TPM actually executes the actions. In such a case the actuator receives

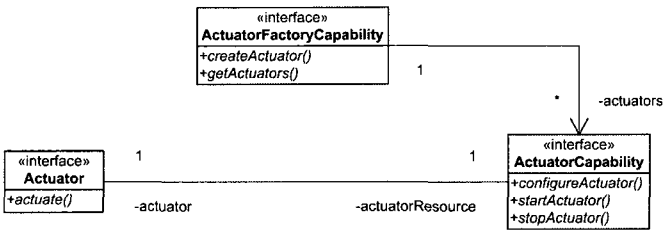

FIGURE 5.20: Actuator Structure

the decision, and translates it into a command that can be sent to TPM for execution by doing a SOAP request to execute a specific workflow. Once the workflow returns the results, the result is packaged as a modified object data structure and passed to the adaptor. Similar to the model and the decision maker, an actuator must implement only one method - actuate, which receives a decision data structure. Figure 5.20 shows the structure of the actuator component:

ActuatorFactoryCapability which is responsible for creating the actuator resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. In order to create an actuator resource the capability receives only the type of actuator to create. The factory also provides clients a way to query what types of actuators are available to create.

ActuatorCapability which is responsible for configuring the actuator, and also starting and stopping the actuator. This implements the *Autonomic Resource* mechanisms. Configuring the actuator includes setting the notification client to the coordinator, as well as setting the notification producer endpoint which accepts subscriptions from its own client - the adaptor. The actuator also receives information about how to contact either the resources it manages or the execution engine it uses. For example for TPM, it would receive the location of TPM's web service endpoint as well as the names of the workflows needed to perform its actions.

Actuator which takes the decision action, and receives the results of that action from the underlying resource. The actuator also monitors the execution of the action in order to ensure the correct operation.

Figure 5.21 shows the sequence of events required to create an actuator, start it, and the internal sequence of the actuator for one execution request received from a coordinator. The creation/configuration/start operations are similar to

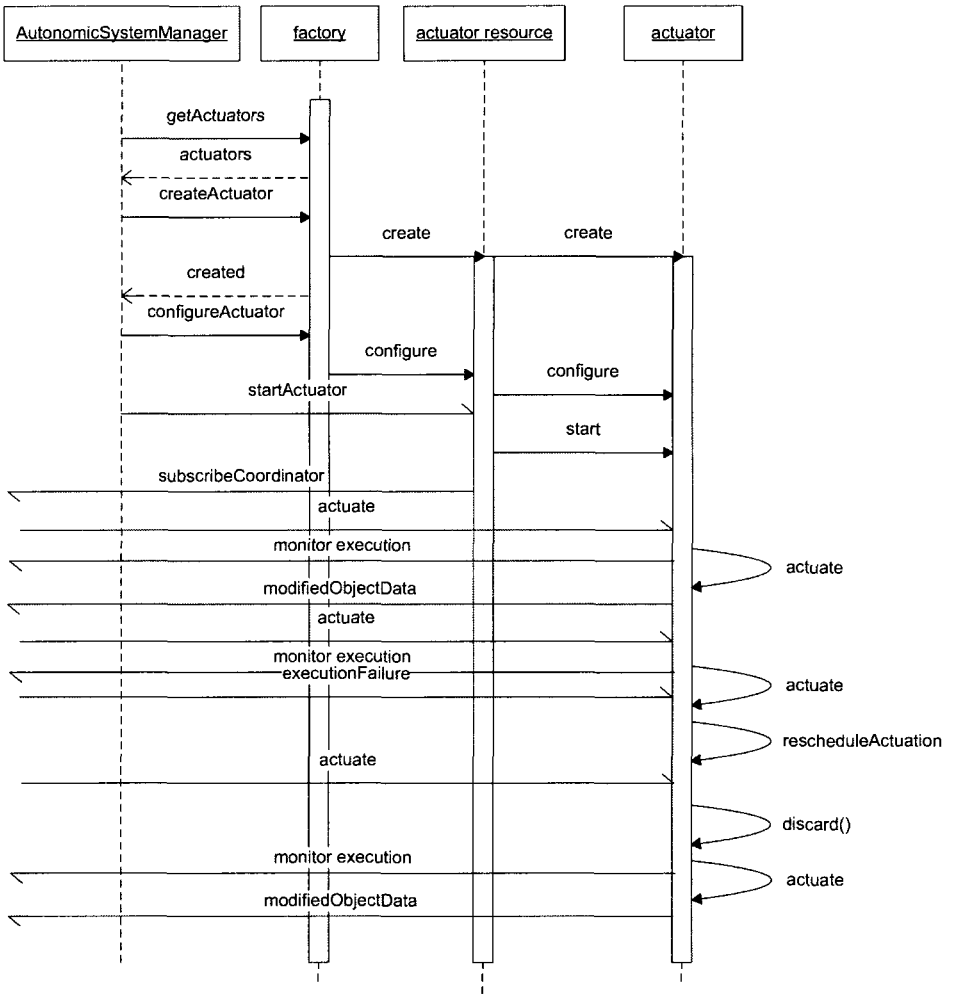

FIGURE 5.21: Actuator Sequence

the model. Once started the actuator receives an actuation request based on a decision made by the decision maker. The actuator then invokes either its own logic to perform the action or communicates with an execution engine. The action can be scheduled for a later date if needed. Two possible results can be achieved - either the actuation is successful and the actuator sends the modified objects to the adaptor, or there is a failure during actuation in which case the actuator can reschedule the action for a later time. The actuator must also be smart enough to determine if a request received at a later time is due to previous symptoms that caused the failed actuation. For example, if the actuator is requested to add a new server to a cluster and the execution fails, the decision maker could, at the next iteration, request a new server again. In such a case, the actuator can detect that both requests are due to the same symptoms and discard one of the requests.

5.1.9 Adaptor component

The adaptor is the component that makes changes in the control loop itself based on changes in the managed resource or in the environment. The existence of the adaptor transforms the autonomic control loop itself into an autonomic system capable of adapting itself. The composition of the adaptor can be a very simple or very complex structure, depending on the reconfiguration that is required. The reconfiguration can range from a simple case where it configures new sensors and ensures that the filter knowns about them, to more complex cases where it modifies parameters at runtime in the other components, to the extreme case where it can actually replace components while the control loop is executing. As such, the adaptor interface specifies a number of functions with different granularities, not all of which must be implemented. Figure 5.22 shows the structure of the adaptor. A control loop can have more than one adaptor, each taking care of different types of adaptations. The sequence of events for the adaptor starts when it receives a modified object data structure from the actuator. In response to the notification, the adaptor decides how this impacts the execution of the control loop, and starts the appropriate adaptation. In order to be able to execute some of its logic, the adaptor must have information regarding how the other components are configured. For example, in order to deploy a new sensor for a new server, it must know how the other sensors are configured in order to have the sensors being homogeneous across a certain metric. In the example described above, adding a new server would result in the addition of a new sensor, and the reconfiguration of the filter, while the removal of a server would result in the removal of a component, and again the reconfiguration of the filter. Replacing a component is defined as a separate method and not simply as a succession of a removal and an addition, as replacement requires part of the control loop to be paused while the replacement is done, while removal assumes no impact on the rest of the control loop's execution. The adaptor is composed of the following parts:

AdaptorFactoryCapability which is responsible for creating the adaptor resource. This implements the *Autonomic Factory Resource* mechanism defined earlier. In order to create an adaptor resource the capability receives only the type of adaptor to create. The factory also provides clients a way to query what types of adaptors are available to create.

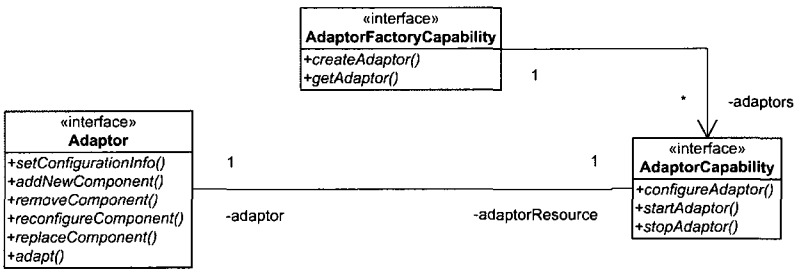

FIGURE 5.22: Adaptor Structure

AdaptorCapability which is responsible for configuring the adaptor, and also starting and stopping the adaptor. This implements the *Autonomic Resource* mechanisms. Configuring the adaptor includes setting the notification client to the actuator. The adaptor also receives information about how to contact the rest of the control loop's elements in order to reconfigure them if needed.

Adaptor which examines changes in the managed resource and takes appropriate action in order to adapt it to the changes. The interface provides a number of functions that can be used for various reconfiguration operations, but the base of the adaptation is the adapt function.

Figure 5.23 shows the sequence of events required to create an adaptor, start it, and the internal sequence of the adaptor for two adaptation requests received from an actuator - an add component and a remove component request. The creation/configuration/start operations are similar to the model. Once started the adaptor receives an adapt request based on a modification to the managed resource made by the actuator. The adaptor then invokes its appropriate method based on the type of modifications received from the actuator - addNewComponent or removeComponent.

5.2 Autonomic Registry Implementation

As mentioned previously, the registry is a key component of the autonomic control loop, as it allows the management of the various components and of the running control loops in the environment. Similar to the way a UDDI registry can be used to find the appropriate web services for a SOA architecture and a management system can bind the services together, the autonomic registry acts as a center

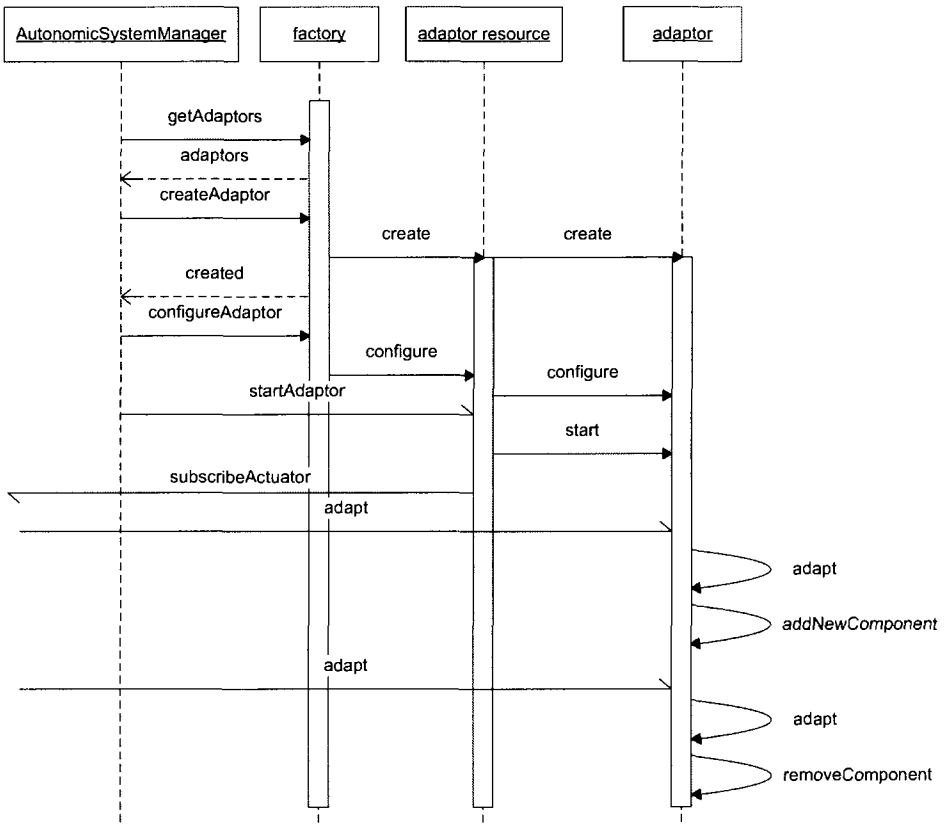

FIGURE 5.23: Adaptor Sequence

point for finding autonomic components which can be composed in a control loop. When an autonomic component container starts, it automatically adds itself to the registry, allowing for future retrieval by a manager. Such a manager could be either a graphical user interface which allows a user to build a control loop, or an automatic script choosing the components that it requires.

Figure 5.24 shows the structure of the registry component. The registry is composed of the three interfaces for registration, retrieval and removal as well as the persistence layer, implemented using the EJB3 specification [51]. The persistence layer is composed of the three entities showed in the figure, as well as the stateless beans used for database access.

RegisterCapability which implements the *Register interface* described in Chapter 4. It provides three functions - registerSystem, registerComponent and

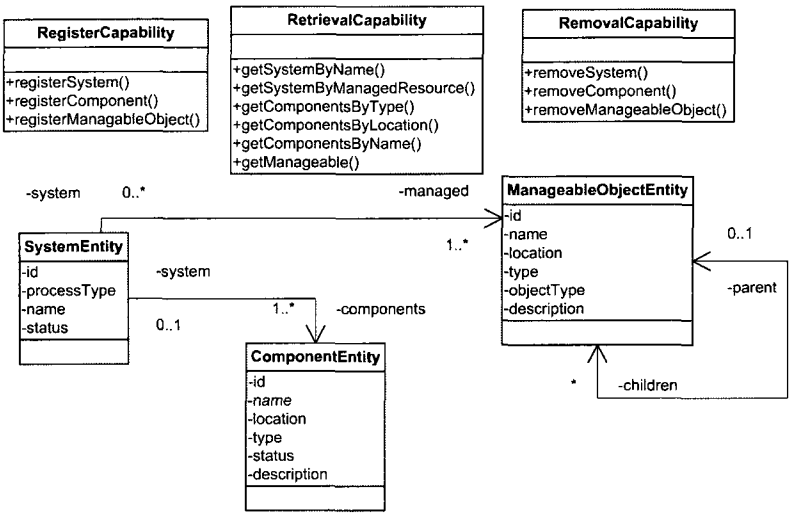

FIGURE 5.24: Registry Structure Implementation

registerManageableObject. Each of the three functions allows the registration of one of the types of data stored in the registry. Registering a component includes using the persistence layer in order to store the data describing the object that is being registered.

RetrievalCapability which implements the *Retrieve interface* described in Chapter 4. The capability allows clients to search for data in the registry, and each of its getter functions return one or more entities.

RemovalCapability which implements the *Remove interface* described in Chapter 4. This capability provides the ability to remove data from the registry. A remove request for one object, based on the object's unique ID, results in the object and its related data being removed from the registry.

SystemEntity which implements the *System* persistable entity. This entity object has as attributes all the data defined in Chapter 4 as well as a unique ID, which is an auto incremented integer. The *ManageableObjectEntity* and *ComponentEntity* object links are represented as *OneToMany* relations.

ManageableObjectEntity which implements the *Manageable Resources* persistable entity. Like the *SysytemEntity* this entity object adds a unique ID. The *children* relation is represented as a *OneToMany* relation, while the parent relation is the inverse *ManyToOne* relation.

ComponentEntity which implements the *Autonomic component* persistable entity. This entity also adds a unique ID, as well as a *ManyToOne* relation to the *SystemEntity*.

5.3 Example Component Structure

One of the main reasons behind the development of the architecture and framework for autonomic computing systems presented in this thesis was to create a system that could be easily expanded by creating new implementations of components. One way of achieving this was by splitting the architecture into components and rooting the components in object oriented patterns. Figure 5.25 shows a possible class hierarchy for the estimator component. The KalmanFilterEstimator and the BayesianEstimator both achieve the same goals in different ways. Each of these two estimators use their own approach in order to estimate the future state of the system, and each have specific limits on systems where the estimation will be valid. Furthermore, due to the object oriented approach used to build the system and the framework's capabilities, each of these two estimators can be further extended - the KalmanFilterEstimator via an ExtendedKalmanFilterEstimator class and an UnscentedKalmanFilter class, and the BayesianEstimator via a RecursiveBayesianEstimator class.

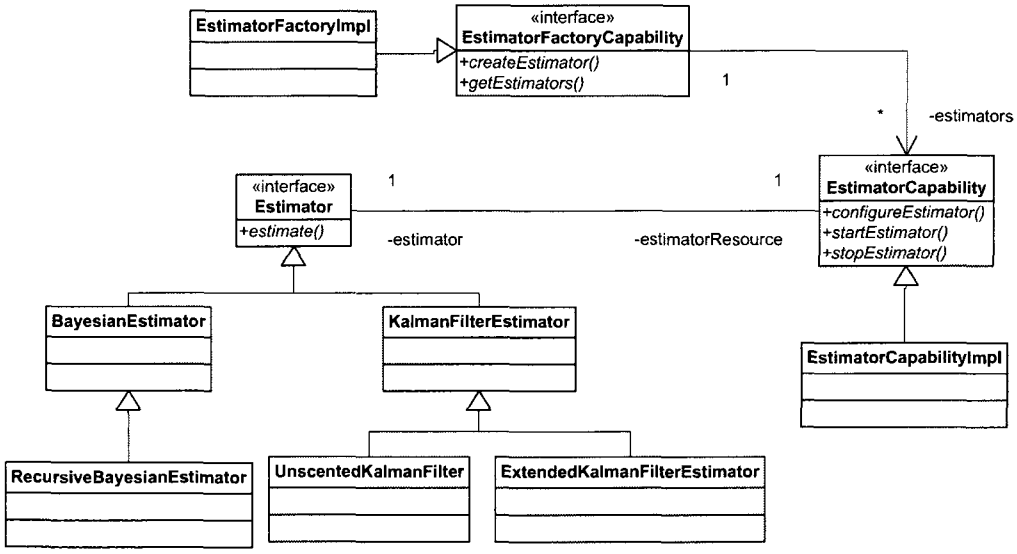

FIGURE 5.25: Estimator Structure Implementation

Note that the class hierarchy could be built differently, as the KalmanFilerEstimator class could extended the RecursiveBayesianEstimator class, and the UnscentedKalmanFilter class could extend the ExtendedKalmanFilterEstimator class.

Through this approach the development of components is simplified, as the ExtendedKalmanFilterEstimator for example can simply modify the internal equations for the estimate while using the same data structures as the KalmanFilterEstimator. At the same time, this component structure simplifies the adaptation of the control loop. When the controlled system's parameters go outside the valid bounds for the autonomic components that are executing, the adaptor can easily replace one component with another especially if they share a similar structure. For example, replacing an Extended Kalman Filter with an Unscented Kalman Filter could be done by simply copying the data structures from the original filter to its replacement and starting the replacement.

Chapter 6

Example Implementations

In order to test the architecture and framework for autonomic system, a number of test autonomic system applications were built. The architecture was first applied to the issue of self-optimization for clusters of application servers. The problem was that of controlling the performance of a cluster of WebSphere application servers, by providing optimal server usage which ensures that the response time of the application is within certain thresholds.

6.1 WebSphere Application Server Cluster Optimization

In such an optimization problem, two conflicting issues are balanced. On one hand, the desire is to use as few servers as possible in order to minimize energy costs and upkeep for the servers. On the other hand the response time of the application servers must be within desired values in order not to impact the application's clients. As such, the problem is one of minimizing the number of servers used, while at the same time maintaining the desired response times. One extra problem appears due to the long time it takes to create new servers. While the desire is to minimize the response time and utilization, actions to remove or add servers do not happen unless they are absolutely needed. For example, if removing a server would improve the utilization but maintain the same response time, but the improvement in utilization is small no action is taken in order not to disturb the system.

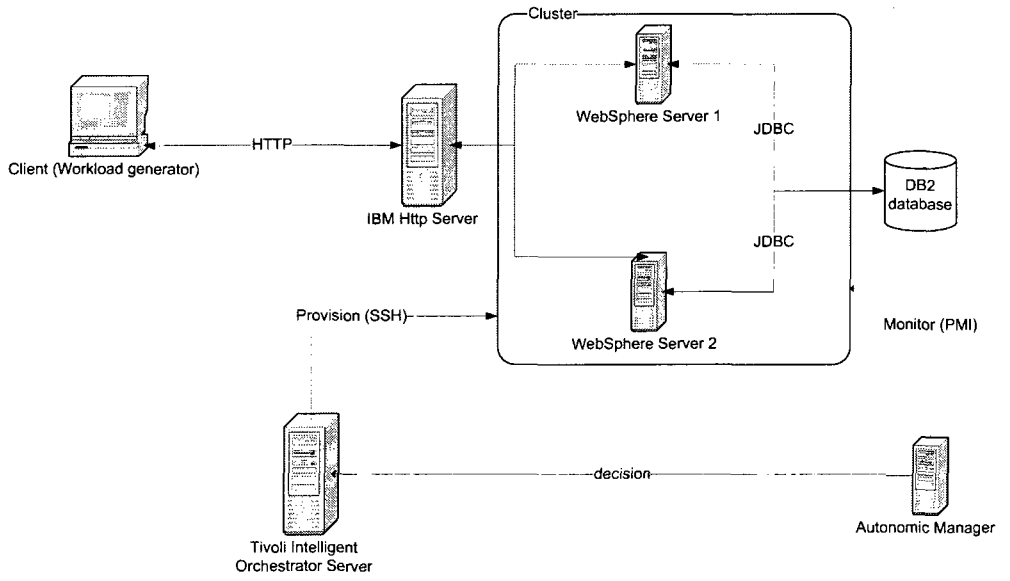

FIGURE 6.1: Testbed Deployment

6.1.1 Test Bed

In order to test the control loop which was built for the optimization of the cluster of servers, a testbed was created which is shown in Figure 6.1.

The testbed was composed of four actual machines, with some machines taking multiple roles from Figure 6.1. The client was a custom built workload generator. The autonomic manager represents the control loop part of the system and TIO was used as the execution engine. All the machines were part of the same LAN and were connected via a 100 Mbps Ethernet connection.

The client machine was a Pentium 4 at 3.4 GHz with 2 GB of RAM. The WebSphere servers were also Pentium 4 machines at 3.00 GHz with 2 GB of RAM. The Tivoli Intelligent Orchestrator machine was a Pentium 4 at 2.4 GHz with 2 GB of RAM. The Tivoli Intelligent Orchestrator machine also contained the DB2 version 8.2 database, while the client machine also executed the autonomic manager. The HTTP server was collocated with one of the WebSphere servers. The WebSphere application servers were clustered through the use of the WebSphere Network Deployment software and were both running the Trade 6 benchmarking application.

The Trade6 benchmarking application is a three tiered web application developed by IBM for benchmarking the performance of WebSphere servers. The application

simulates a stock trading application in which a client can log in via a user name and password, look at the holdings he has, buy or sell stocks and also view quotes for various stocks. The application executes via a number of servlets and each request results in one or more queries to the back end database through the use of Enterprise Java Beans 2.0. The benchmarking application provides a servlet which is used to generates requests in the application that simulate typical user actions. These actions are semi random, such that two threads sending requests will generally simulate different user patterns.

The HTTP server acted as a load balancer in front of the Application Servers. All requests were sent to the HTTP server, and the server redistributed the requests to the two WebSphere servers using Round Robin load balancing. The workload generator was a Java program that created a number of threads each loading an URL - the simulate servlet URL. Each thread slept between consecutive requests, and the application started a set number of threads based on a user distribution. In some test cases, the user distribution was set to change dynamically and as such client threads were started and stopped dynamically in order to preserve the required distribution. The TIO execution environment executed its own workflows in order to add or remove servers, and part of the workflow execution included using SSH and SFTP connections to the servers to copy files and to execute remote commands on the servers.

6.1.2 Autonomic Components

The autonomic control loop which was created for management of the control loop implemented each of the eight components described previously in the architecture.

6.1.2.1 WebSphere Sensor

The sensor for the WebSphere Application Server is based on the JMX data provided by the application server. The WebSphere sensor is deployed as part of the WebSphere server it is monitoring. When the server starts, the sensor's factory capability is also started and initialized, and the sensor registers itself with the autonomic registry. For a cluster of WebSphere servers it is possible to have multiple server instances deployed on the same machine. This is achieved by deploying the server in two components - a nodeagent which acts as a unique manager for the

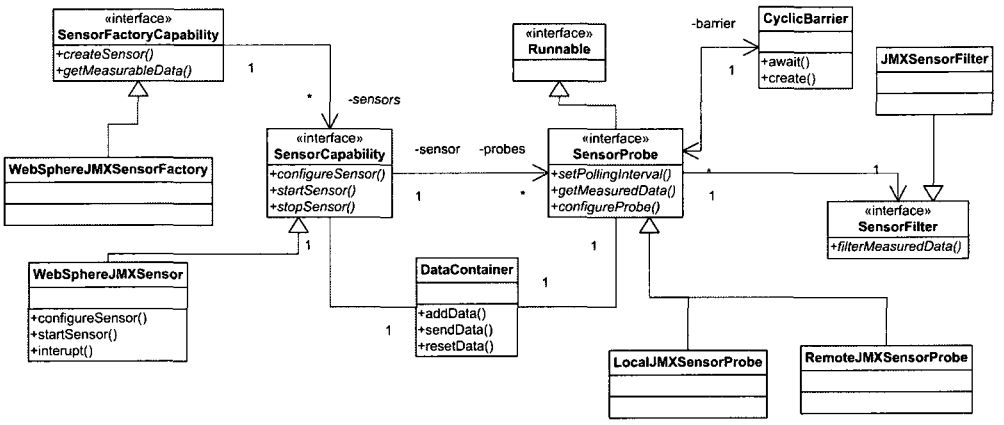

FIGURE 6.2: WebSphere Sensor

servers deployed on the same machine with it and is capable of starting/stopping the server instances and a number of separate server instances. This is important because it affects the sensor's design. Figure 6.2 presents the structure of the WebSphere sensor and how the interfaces previously described are implemented.

WebSphereJMXFactory - the factory implements the *SensorFactoryCapability* interface. First of all, the factory communicates with the server's JMX configuration in order to determine what metrics can be measured by the sensors. The factory also provides a way to create sensors for the WebSphere instance it is deployed in.

WebSphereJMXSensor - the sensor implements the *SensorCapability* interface. The sensor configures its internal structure and sets up its appropriate probes in order to measure what a client is interested in from the WebSphere server.

JMXSensorFilter - the sensor filter implements the *SensorFilter* interface. The sensor filter knows how to receive data measured via JMX and remove unnecessary data, based on a configured list of required data.

LocalJMXSensorProbe - the local probe is meant to gather data via JMX from a local WebSphere instance. It connects via the local JMX client defined by IBM for WebSphere servers.

RemoteJMXSensorProbe - the remote probe is meant to gather data via JMX from a remote WebSphere instance. It connects via RMI to a remote WebSphere server in order to gather data. The goal of the remote probe is to

gather data from a nodeagent regarding the hardware the server is running on.

For the cluster optimization problem presented in this section the autonomic system is interested in measuring the following data from the servers:

1. CPU utilization of the server. This value is obtained from the nodeagent via the *SystemMetrics* JMX object and provides the average CPU utilization since the previous measurement.

2. Request count for the Trade6 scenario servlet. This value is obtained from each of the servers via the *Servlet* JMX object and represents a counter for the total number of requests received by the servlet since the application server started.

3. Total service time for the Trade6 scenario servlet. This value is obtained from each of the servers via the *Servlet* JMX object and represents the total time taken by the application server in order to service requests received for the servlet.

When a new sensor is created for a WebSphere server, the sensor creates its two probes - the local probe monitors the *Servlet* JMX object and filters the data coming from the JMX system in order to preserve only the request count and total service time attributes. The remote probe monitors via RMI the nodeagent for the server and retrieves data regarding the CPU utilization of the machine. The monitoring is done by the sensors in an active manner - the sensors periodically request the JMX system to do a measurement and are not based on notifications by the JMX system. For the problem presented in this section, the sampling time is set to 30 seconds.

6.1.2.2 Filter Manager

The filter for the given problem is based on the data being measured and data that is required by the rest of the control loop. The following measurements are what the control loop expects in order to be able to optimize the application server cluster:

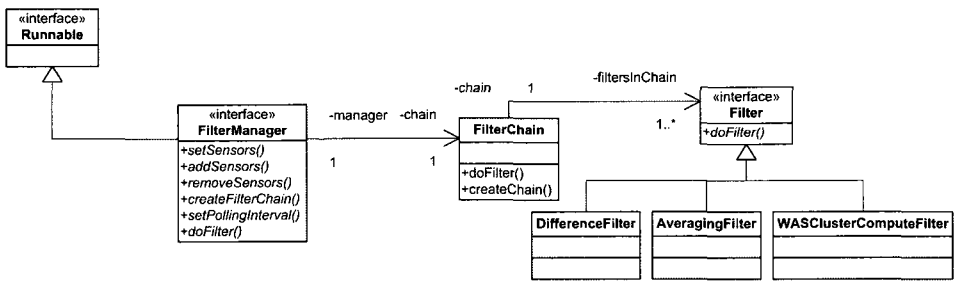

FIGURE 6.3: WebSphere Filter Manager

1. Average CPU utilization - the average of the CPU utilization across all servers. This is simply computed as $\frac{\sum_{i=1}^{N} CPU_{utilization}}{N}$ where N is the number of servers in the cluster and $\text{CPU}_{utilization}$ is the measured CPU utilization for each server.

2. Average number of users - the number of users that made requests to the scenario servlet in the last sampling time frame. This is computed as $\frac{\sum_{i=1}^{N}(Users_n - Users_{n-1})}{N}$ where $Users_n$ is the number of users measured at time n and N is the number of servers.

3. Average response time - the average response time of the scenario servlet. This is computed as $\frac{\sum_{i=1}^{N} \frac{ServiceTime_n - ServiceTime_{n-1}}{Users_n - Users_{n-1}}}{N}$ where $ServiceTime_n$ is the measured service time at time n, $Users_n$ is the number of users measured at time n and N is the total number of servers.

4. Average throughput - the average number of user requests processed per second. This is calculated as $\frac{\sum_{i=1}^{N} \frac{Users_n - Users_{n-1}}{\Delta t}}{N}$ where $Users_n$ is the number of users measured at time n, N is the total number of servers and Δt is the sampling time.

Since a number of the calculations have common parts, the calculations are split into three simple filters as shown in Figure 6.3.

The three filters all implement the Filter interface and the interface's single method: *doFilter*.

DifferenceFilter The DifferenceFilter receives two types of input - the current values measured by the sensors and the values from the previous measurement. Based on these inputs the filter computes $(Users_n - Users_{n-1})$ and $(ServiceTime_n - ServiceTime_{n-1})$. All other values are left unchanged.

WASClusterComputeFilter This filter computes the values required by the control loop which are not measured directly by the sensors - the response time and the throughput. Based on the data computed by the previous filter in the chain, the *DifferenceFilter*, the *WASClusterComputeFilter* calculates for each server $\frac{\Delta ServiceTime}{\Delta Users}$ and $\frac{\Delta Users}{\Delta t}$.

AveragingFilter The averaging filter computes the average of the data points across all the servers. For all of the input values, which are based on the data computed by the *WASClusterComputeFilter*, it calculates $\frac{\sum_{i=1}^{N} Value}{N}$.

Due to the way the filter computations are split, both the difference filter and the averaging filter can be reused for other control loops, since the doFilter functions that they implement are general and do not distinguish their computations based on the input data.

6.1.2.3 Iterative Convergence Coordinator

The control loop logic used to solve the self-optimization of a cluster of servers is based on Queuing Model Analysis similar to the solution presented in [52]. In such a control system, the estimation of the future state of the system is computed multiple times for each measurement of the system. At each iteration, the model is updated, the future state is estimated, and the system checks whether two consecutive estimations are within a certain distance from each other and thus converge. If two consecutive values converge the latest value is considered the appropriate estimate and the iterative loop breaks. If after a certain number of iterations there is no convergence, the system gives up and uses the latest value as the appropriate estimate. Figure 6.4 shows the sequence of events in one coordination iteration.

6.1.2.4 Layered Queuing Model

The model used for representing the state of the system is a layered queuing model. The queues of the system's model represent locations where the requests are waiting to be processed or locations where the requests are actively being processed. Figure 6.5 shows the structure of the queues. In a three tier system like the one described in this section, the client sends a request to the HTTP

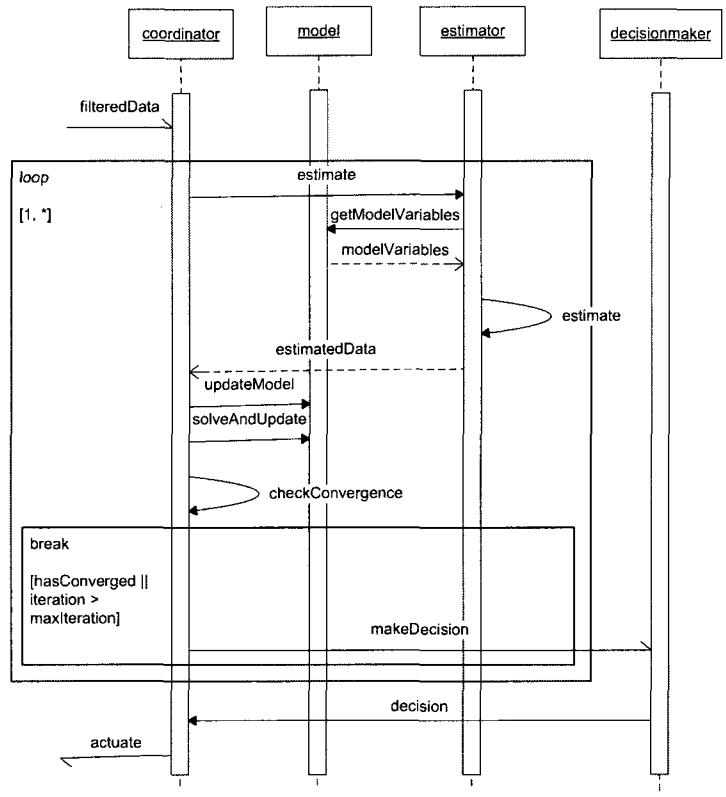

FIGURE 6.4: Iterative Convergence Coordinator Sequence

server, where the request waits to be redirected to one of the application servers. Once at the application server, the request waits for a free thread in order to be processed, is processed - which can include a request to the database server, and waits to be sent as a reply. The Layered Queuing Model used only models the interaction at the application server level, which includes the database server request as part of itself. This can cause provisioning problems if the database server is the bottleneck, however for the testbed presented in this section the database is never the bottleneck. The HTTP server queue times are considered constant and insignificant to be modeled. All the servers in the cluster are modeled together as one server, and as such the average of the cluster's performance represents the performance of this "global" server.

The data modeled is the same as the data described in the filter section for the filter's outputs - response time, number of users, utilization and throughput. In order to solve the model a special package built by IBM was used named Application Performance Evaluator and Resource Allocation Tool (APERA) [53]. The

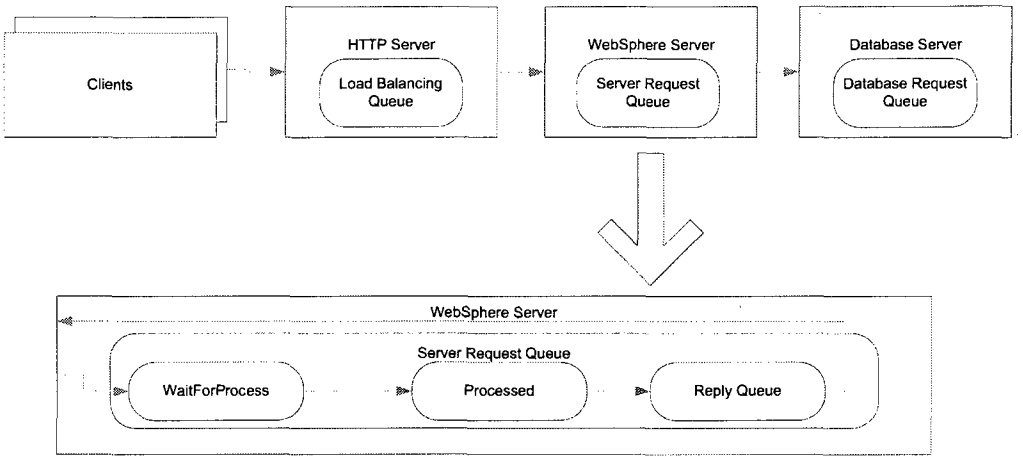

FIGURE 6.5: Layered Queueing Model - 3 Tiers

APERA software is a Java package which allows the creation of an LQM description via an XML file, and then can be used to solve and update the model.

6.1.2.5 Kalman Filter Estimator

The estimator used is based on Kalman Filtering techniques [25]. The goal of the Kalman Filter is to estimate the future state of a discrete-time controlled system which is governed by a linear equation. The input for the future prediction of the system's state is based on an observation vector, obtained by combining the layered queuing model and the latest measured values. The Kalman Filter execution happens in two steps:

1. Time update, which is the prediction stage. At this stage the Kalman Filter predicts the future state of the system and the covariance matrix for the estimate using the following two equations:

$$\widehat{x}_k^- = A\widehat{x}_{k-1} + Bu_{k-1} \tag{6.1}$$

$$\widehat{P}_k^- = AP_{k-1}A^T + Q \tag{6.2}$$

2. Measurement update, which is the Kalman Filter correction stage. At this stage the filter corrects its internal structure based on the measured values, or in this case the LQM's modeled values. This is done by first computing the Kalman gain, followed by updating the state and covariance matrix as

shown in the next three equations:

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1} \tag{6.3}$$

$$\widehat{x}_k = \widehat{x}_k^- + K_k(z_k - H\widehat{x}_k^-) \tag{6.4}$$

$$P_k = (I - K_k H) P_k^- \tag{6.5}$$

Whenever an estimate request is received by the Kalman Filter Estimator, it performs the measurement update stage based on the measured/modeled data followed by a time update stage which predicts the future of the system. As the estimate step is performed multiple times by the iterative coordinator, the Kalman filter will perform multiple measurement updates based on the model data which is updated via its time update function.

6.1.2.6 Threshold Based Decision Maker

The decision maker used is a simple threshold based optimizer. The goal of the autonomic system is to maintain the SLA of the cluster of servers in terms of minimum response time, while at the same time using a minimum number of servers. In order to achieve this, the decision maker uses two thresholds - a minimum and a maximum response time threshold. As long as the estimated response time is within the given thresholds, no decision is taken in order to change the managed resource. If the estimated response time goes outside its bounds, the decision maker attempts to determine what would be the minimum number of servers that would bring the response time back between the thresholds. The optimization problem of finding the minimum is done via a binary search over the model space. The decision algorithm can be formally described as follows:

Algorithm 6.1 Decision Maker Pseudocode

```
if LowerThreshold ≤ EstimatedValue ≤ HigherThreshold then
  do nothing;
else
  if LowerThreshold > EstimatedValue then
    findMinimum(minServers, currentServers)
  else
    findMinimum(currentServers, maxServers + 1)
  end if
end if
```

Where minServers represents the minimum number of servers the cluster should contain, currentServers is the current number of servers in the cluster and maxServers is the maximum number of servers the cluster can contain. The findMinimum process is a recursive processes and is described bellow:

Algorithm 6.2 findMinimum (min, max) Function

```
newServerValue := ⌊(min + max)/2⌋
if newServerValue == min then
  return newServerValue
end if
update model with newServerValue
solve model
if LowerThreshold ≤ newResponseTime ≤ HigherThreshold then
  newServerValue := findMinimum (min, newServerValue)
else
  if LowerThreshold > newResponseTime then
    newServerValue := findMinimum (newServerValue, max)
  else
    newServerValue := findMinimum (min, newServerValue)
  end if
end if
resetModel
return newServerValue
```

Once a decision on whether to add or remove servers is taken, the decision maker also chooses which servers to add/remove.

6.1.2.7 TIO Actuator

The actuator used to deploy new application server replicas and add these replicas to the cluster does not perform the actions itself, but rather uses an external TPM deployment which is part of a TIO installation in order to perform the appropriate actions. Once a request to add or remove a certain number of servers is received by the actuator, the actuator packs the request in a format accepted by TPM and sends a request to TPM to execute a workflow with the given parameters which include the cluster to add/remove the server to/from, and the servers to add/remove. The decision of which servers to add/remove is taken by the decision maker and passed to the actuator.

In order to communicate with TPM, the actuator uses the Web Service interface defined by IBM for TPM. While the workflow is being executed by the TPM

instance, the actuator periodically checks the status of the execution. Once the execution has succeeded the actuator retrieves the result of the execution, which contains the servers added or removed and packs them into an array of *ModifiedObject* data structures which is returned passed to the adaptor. If the deployment fails, the actuator requeues it for later deployment up to a certain number of times. If too many failures happen, the actuator notifies the adaptor which can stop the control loop if needed.

While executing the request the actuator maintains information about the state of the cluster before the request was sent, and ensures that if other deployments are received while it is executing the cluster does not go into a conflicting state. For example, if the cluster has two servers, the actuator receives a request to add a new server at time t, and later at time t + x it receives again a request to add a server, the adaptor will ensure that the old request has completed successfully. If the old request has not completed, the actuator will discard the new request. Similarly, if the second request asks the actuator to remove a server, the actuator will assume that changes in the environment have caused this request. Because of this, the actuator will ask TPM to stop the previous request and instead remove a server. If the previous deployment of adding a server has just finished, the actuator will remove two servers instead of one.

6.1.2.8 Reconfiguring Adaptor

The adaptor used is only capable of reconfiguring the components in the control loop, and is not capable of adding or removing components from the control loop. Based on a list of *ModifiedObjects* received from the actuator, which contains the names of the servers added or removed, the adaptor performs the following actions:

1. Configure new sensors based on the configuration of the sensors already existing in the systems. The adaptor connects to each of the newly created servers, and based on the information it has about how the other sensors are configured, it configures and starts these sensors.

2. Reconfiguration of filters in order to add the new servers. Once the new sensors have been configured and started, the adaptor notifies the filter that new sensors exist to which the filter should subscribe.

3. Reconfiguration of the model. The model is reconfigured with the new number of servers in order to ensure that the model has the appropriate knowledge.

6.1.3 Administrator User Interface

A web based user interface was also built for the system, in order to allow an administrator to easily create and deploy a control loop. The user interface also allows an administrator to monitor the execution of an already created control loop. The user interface interacts directly with the registry in order to find what components are available and what systems have already been deployed. Figure 6.6 shows the layout of the graphical user interface.

The user interface presents a manager of an autonomic system, a guided way to create an autonomic computing solution. For each of the components to be created for the autonomic control loop, the user is presented a wizard in order to configure the component. The information regarding what is to be configured for each component is retrieved directly from the component via WSDM's metadata capability. The left side of the UI contains a tree with available components found via the registry, the user can then drag and drop items from the left side tree to the component holders displayed in the right side loop.

The first step in creating an autonomic control system, is to define what type of container is to be managed. The top right hand side box allows the administrator to configure this. In figure 6.6, the administrator has the option of controlling a WebSphere Application Server cluster, a Virtual Machine cluster or a DB2 cluster. Based on the choice made by the administrator, the component tree is pruned to remove components that can not be used for the chosen container type. Figure 6.7 shows the two wizards which form the container selection process. On the left hand side is the wizard where the user selects the container type to be managed. Once a container type is selected, the second tab becomes active and the user can select which instance in the datacenter to manage. The second tab is populated based on the choice selected in the first tab. In the figure, only one WebSphere cluster is available which is named TradeCluster and whose deployment manager (Dmgr) resides on the server named cerast10.

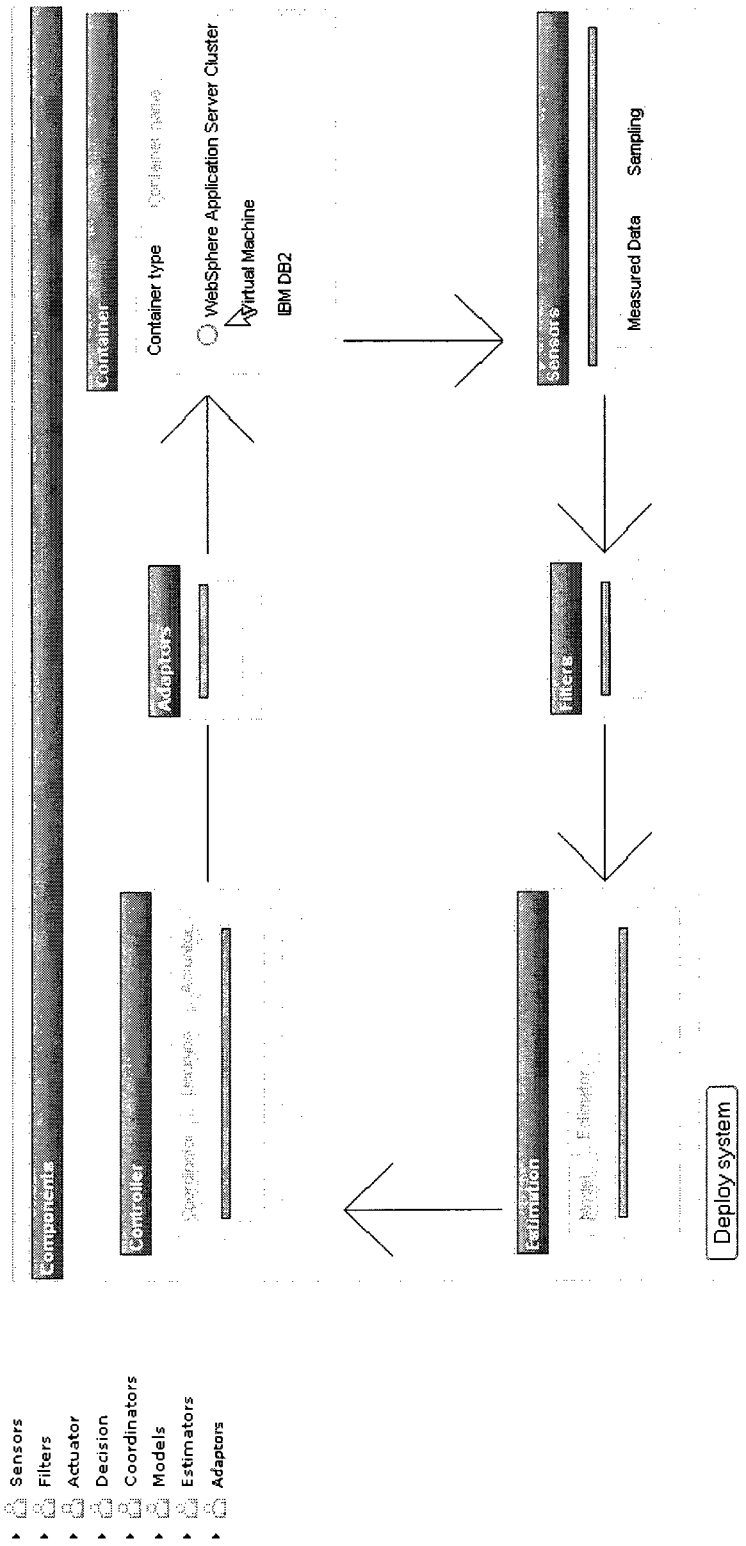

FIGURE 6.6: GUI of the Autonomic Computing Solution

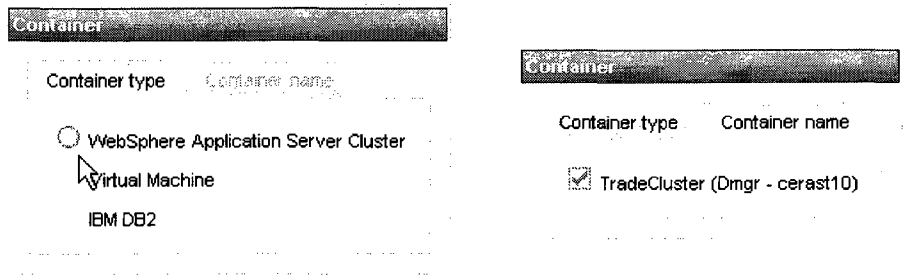

FIGURE 6.7: Container Configuration

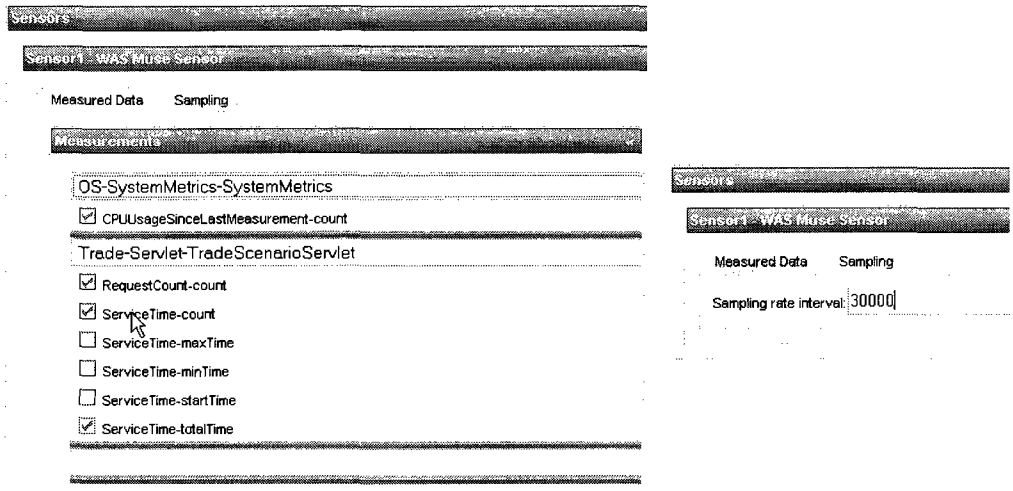

FIGURE 6.8: Sensor Configuration

After the container to manage has been selected, the administrator can configure the sensor to use in order to measure the required data. The user first drag and drops the appropriate sensor on the *Sensors* area in figure 6.6, in this case a *WAS Muse Sensor*, and then configures the sensors. In this example it is assumed that all the sensors on all the servers are capable of providing the same data. As such, the data regarding what can be measured by the sensors comes from one of the servers in the cluster chosen randomly. The sensor configuration is done in two steps - first the user chooses what data to measure, and then he selects the sampling interval for the sensor. Figure 6.8 shows the sensor configuration process. The measured data configuration tab presents the data as represented in JMX. The user can select from a drop down what object to retrieve data from, and after that he can select which variables of that object to measure. The drop downs result in one probe created per drop down, and the check marked variables result in data that is not removed by the sensors' filtering structure.

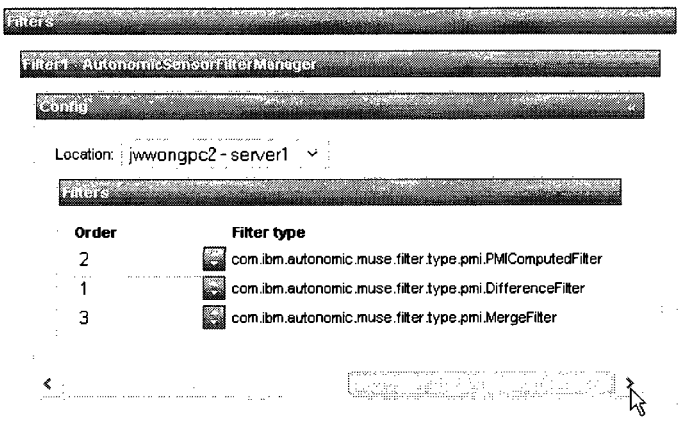

FIGURE 6.9: Filter Configuration

After the sensor configuration, the filter configuration follows. First the administrator chooses the filter manager type to use by drag and dropping it on the *Filter* area in figure 6.6. Once a filter manager type is selected, the user can select from a drop down the location where the filter manager should be deployed. This choice is based on the machines in the data center that have registered themselves as having the respective filter manager as available. After the manager is selected, the filter manager factory is contacted in order to find the filters which reside at the endpoint. A list of these filters is displayed and the administrator can choose the order of chaining the filters. Filters with a value of 0 are not used. Figure 6.9 shows the filter configuration wizard. This filter is an *AutonomicSensorFilterManager* deployed on the jwwongpc2 host and instance server1 of WebSphere Application Server on that host. The three filters that are chained, in order, are the *DifferenceFilter*, the *PMIComputedFilter* and the *MergeFilter*.

After the filter, the model is configured. The model and the estimator are placed together in the GUI in a container named estimation in order not to clutter the UI. Like the filter, the administrator chooses the type of model to use - a *PMIModel* in this case, the location where the model should be deployed - the constanta host machine, and the model configuration. In this example, the full model is preconfigured for the management of the WebSphere cluster, and only the mapping between the model's internal structure, and the data coming from the filter is given by the autonomic administrator, as shown in figure 6.10.

The estimator configuration is also very simple. Like the other components, the

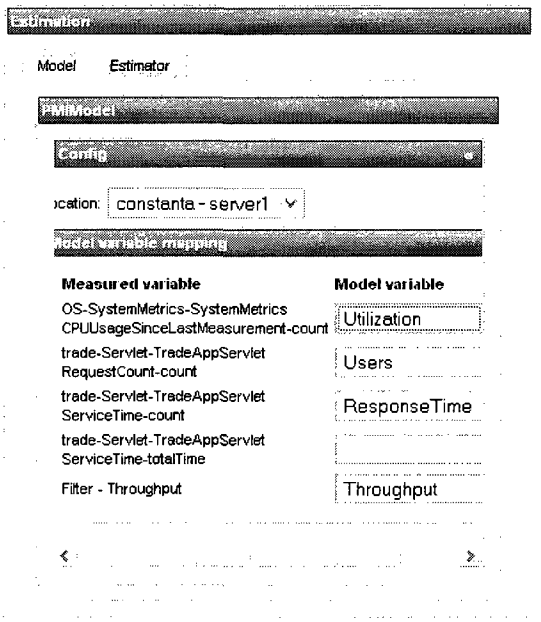

FIGURE 6.10: Model Configuration

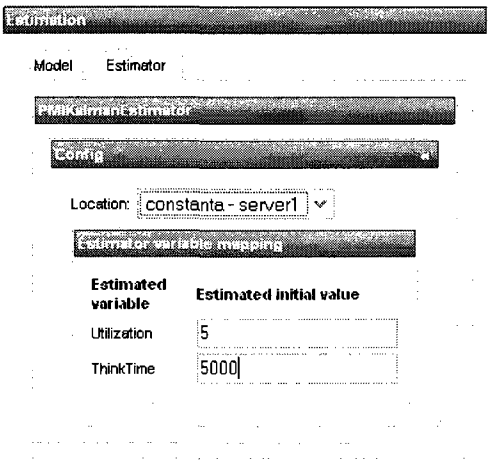

FIGURE 6.11: Estimator Configuration

autonomic administrator chooses the type of estimator to use - a *PMIKalmanEstimator*, and the location where the estimator should be deployed which is the same as the model. The configuration of the actual component consists of setting the initial values of the estimated parameters which are the utilization and the think time. The rest of the configuration - the number of parameters and the number of variables are set by default without user input.

Again, in order not to clutter the UI, the coordinator, decision maker and actuator are all placed together in a container named Controller. All three components are

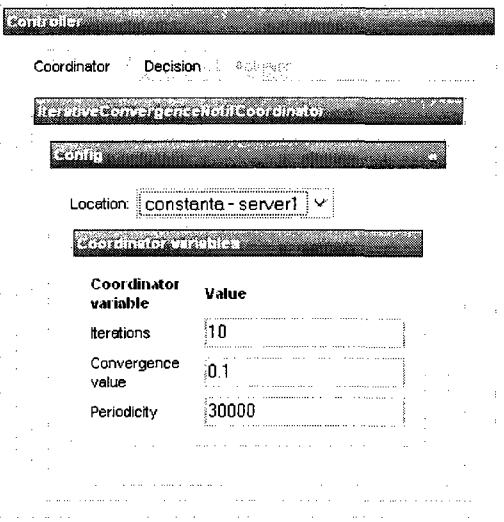

FIGURE 6.12: Coordinator Configuration

similarly configured by choosing the type of component and the deployment location. All three are deployed on the constanta node and the types of components are - *IterativeConvergenceNotifCoordinator*, *PMIDecision* and *TPMExecutorActuator*. The coordinator configuration consists of setting the maximum number of iterations - 10, the convergence value - 0.1 and the periodicity, which represents how much time to wait between two coordination actions - 30000 milliseconds. The decision configuration consists in setting the upper and lower thresholds for the controlled variable - 25 milliseconds and 20 milliseconds respectively for the response time. Finally, the actuator configuration consists of setting the location of the machine where TPM is running on, the port where TPM receives incoming requests as well as the user name and password of a TPM user who can execute workflows. Figures 6.12 through 6.14 show the three configuration windows. The adaptor configuration consists only from choosing the type of adaptor and location to deploy, and will not be shown.

Once all the components have been configured in the GUI, the administrator can deploy the system. The deploy command, sends the appropriate create and configure commands to each of the components. Once all have been created and configured the user can start the system, which sends a start command to each of the components. The deployed components and system are saved in the registry and later the user can change the view and see the performance of the running system.

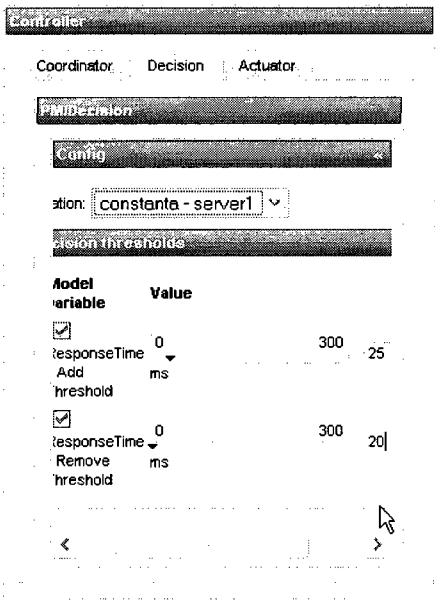

FIGURE 6.13: Decision Configuration

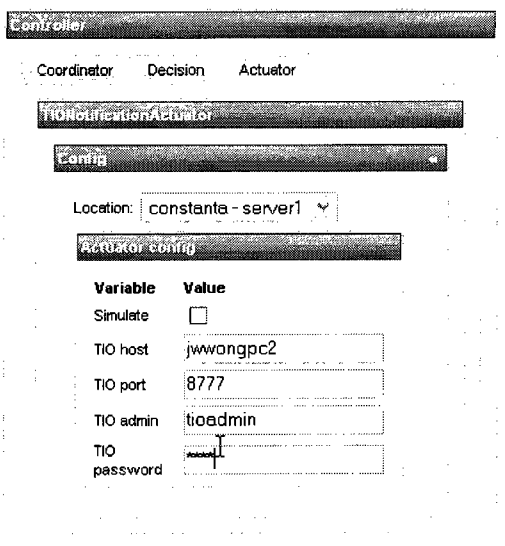

FIGURE 6.14: Actuator Configuration

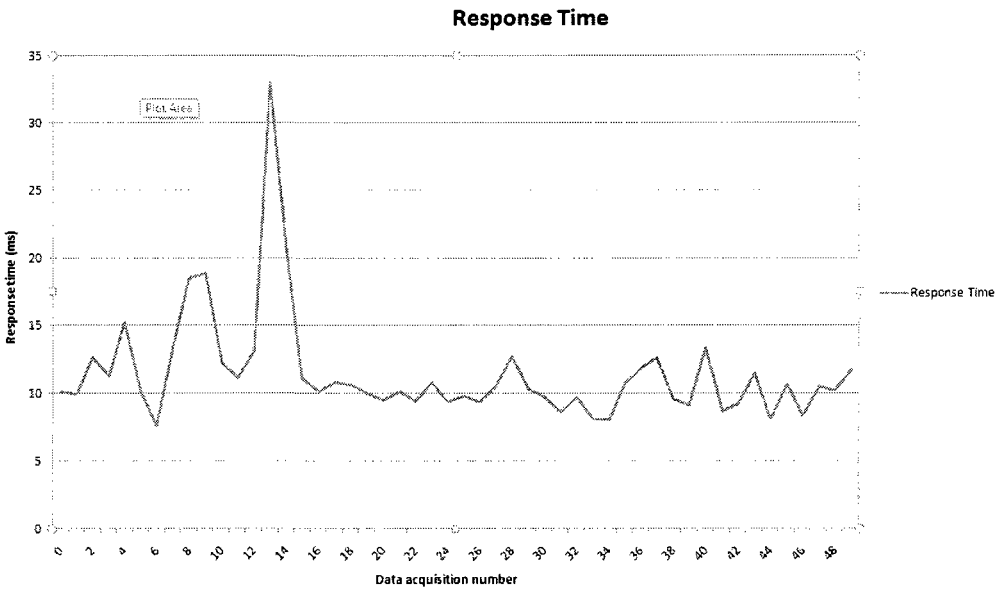

FIGURE 6.15: Response Time - Low Demand

6.1.4 Results

The results presented here are meant to show the effectiveness of the proposed architecture. They do not focus on how the model and estimator perform in order to predict the future of the system as this has been shown in various research papers [52]. The results included next demonstrate why a distributed application works better than a single monolithic one. This is achieved by simulating a large cluster of servers and increasing quickly the number of users making requests to the server. In order to simulate a steadily increasing load on the cluster ten new client threads are launched every minute, up to a total of 120 clients. At the same time, in order to simulate a large cluster of servers and show how the monitored data would change while the data is gathered iteratively from the servers, we loop multiple times and retrieve the data from the same server. In the shown results the necessary data is collected 50 times from the same server while the demand steadily increases. The time necessary to retrieve the data from one server is approximately two seconds. Two sets of images are shown, one for the low end of the demand on the server, with less than 40 clients, and one for the high end of the demand, where there are between 90 and 110 clients.

What can be seen from these figures is that an iterative polling mechanisms where a central unit periodically gathers data iteratively from its sensors does not work

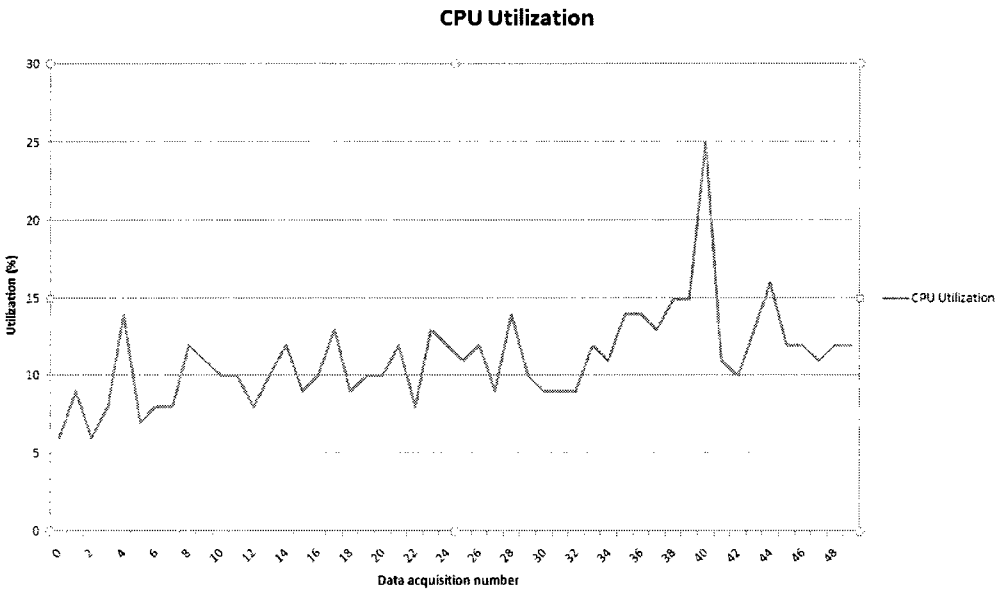

FIGURE 6.16: CPU Utilization - Low Demand

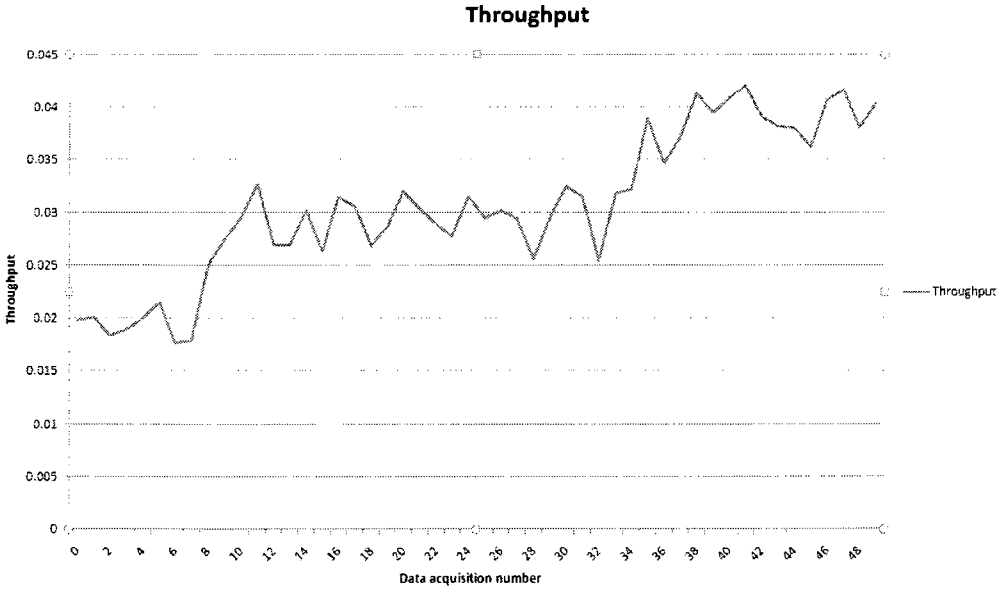

FIGURE 6.17: Throughput - Low Demand

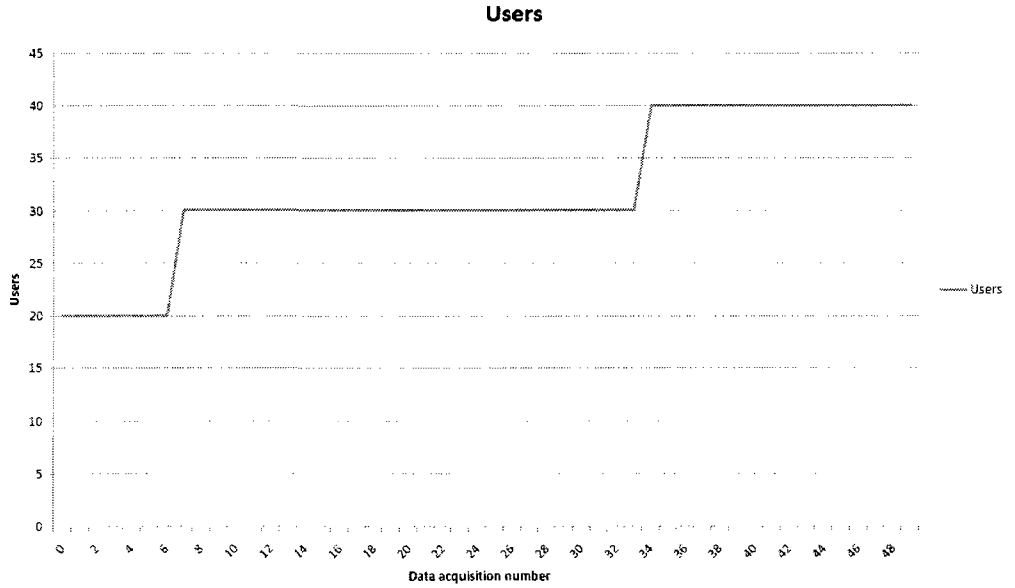

FIGURE 6.18: Users - Low Demand

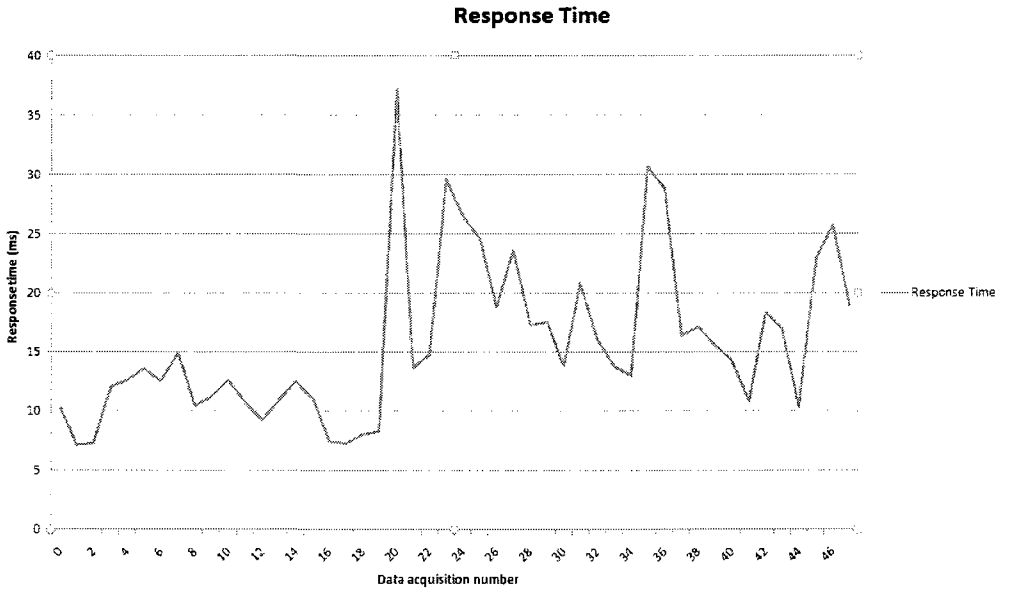

FIGURE 6.19: Response Time - High Demand

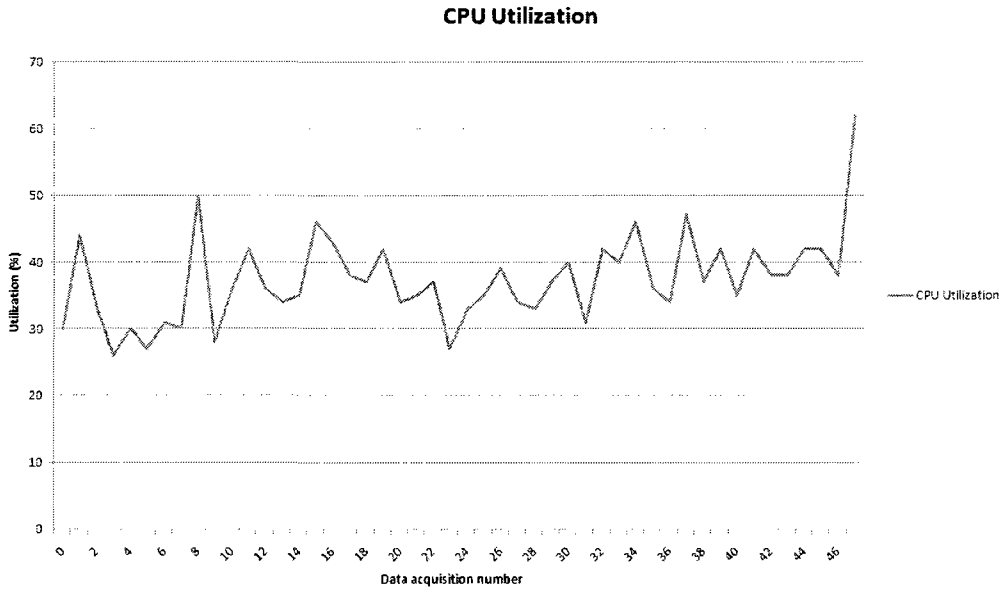

FIGURE 6.20: CPU Utilization - High Demand

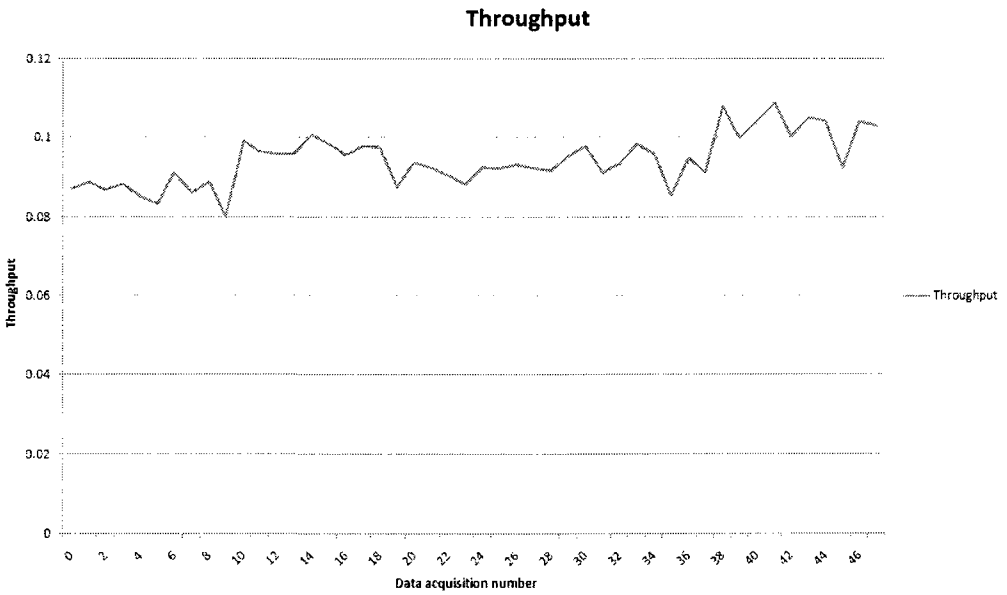

FIGURE 6.21: Throughput - High Demand

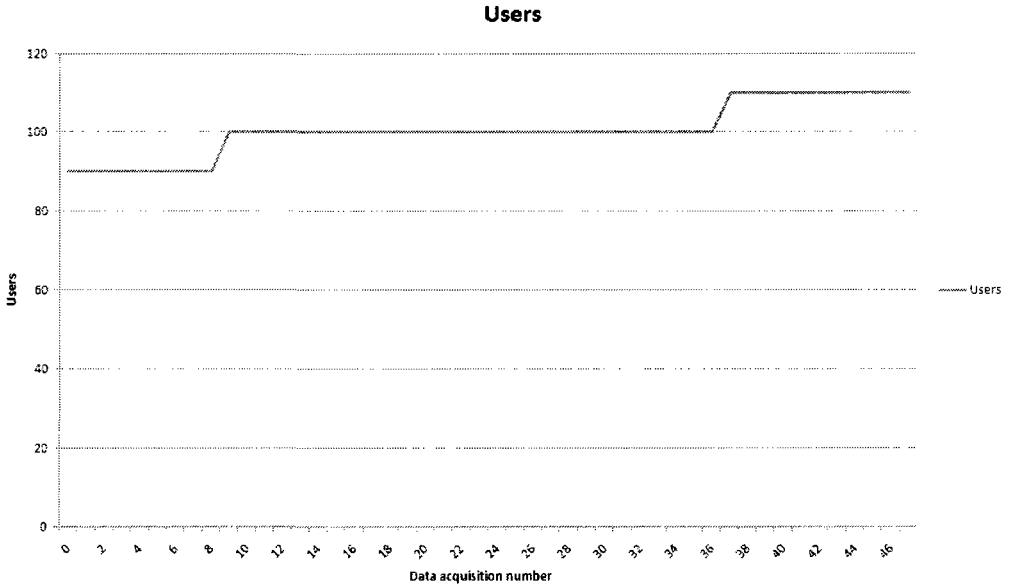

FIGURE 6.22: Users - High Demand

in an environment that changes heavily. By the 50^{th} iteration, in both cases, the CPU utilization nearly doubles. As such, by the time an iterative mechanism would reach the last server in a large cluster, the data from the first server would already be out of date. While the CPU utilization graph is the most pronounced, the other graphs exhibit similar characteristics in that the data would be out of date.

6.2 Virtual Machine Optimization

A different optimization problem for autonomic computing is the problem of optimizing the distribution of virtual machines across hardware machines in a data center. The problem is similar to the one presented before, where a cluster of WebSphere servers is to be optimized. Unlike the previous example where each WebSphere server had its own machine, in this example each WebSphere server resides on top of a Xen Virtual Machine [54]. In order to optimize the data center, the autonomic system makes decisions regarding which new VMs have to be deployed or how to modify the resources used by existing VMs. The decision of adding resources is reached based on a number of factors.

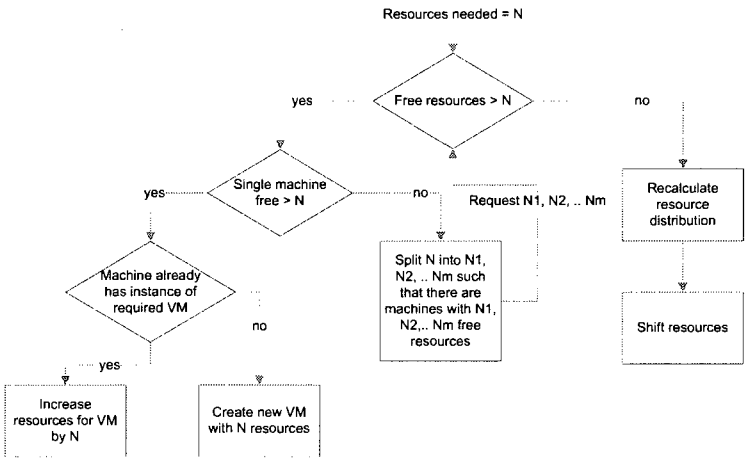

FIGURE 6.23: VM Decision Making

If there are enough free resources in the data center to create a new VM, the decision of what to do is based on where the free resources are located. If there are enough free resources on a machine that already has a VM which is running the application needing more resources than the exiting VM is given more resources. If no such machine exists, than a new VM is created on a machine which has enough unused resources. In the case where none of the above two can be accomplished, the system calculates a way to add resources to an existing VM and create a new VM with the remainder resources. The approach of modifying the CPU usage of various VMs is similar to the idea in [55]. If there are not enough resources in the data center, then the system recalculates the resources required by each application, and shifts the resources in such a way as to provide the best possible response time for all applications based on how critical each application is.

In order to create this autonomic system, part of the previous solution for a cluster of WebSphere servers can be used with no changes. Due to the modular aspect of the architecture and through the use of the GUI, old and new components can be combined to create a new autonomic system with a different goal. The following changes were made to the previous control loop:

Virtual Machine Sensor - This sensor provides information directly from the guest VM in terms of CPU utilization and memory allocation of the VM.

VM Decision Maker - The decision maker decides what action to take, as described before. Figure 6.23 shows the logic for decision making.

VM Actuator - Two actuators are used - the first actuator is responsible for creating new VMs and removing VMs when they are no longer needed. This actuator is also responsible for configuring the VMs and starting the application server as well as any other necessary applications. The second actuator takes care of redistributing the CPU time and memory allocated to each VM.

VM Adaptor - On top of the responsibilities of the *Reconfiguring Adaptor*, this adaptor switches seamlessly between two decision makers based on the free resources in the cluster. When there are available free resources, the decision maker used simply decides where to create a new VM and how much of the underlying system's resources the VM will take. The same decision maker decides when to free up VMs that are no longer needed by the applications. The second decision maker is used to decide how to repartition the used resources when no free resources are available, or when insufficient free resources are available. The same decision maker repartitions resources when a VM's resources are released, if the system was previously strained for resources.

The virtual machine optimization was tested on the same testbed as the WebSphere optimization example but with Xen VMs running between the hardware and the application servers. Figures 6.24 through 6.27 show the average response time, the average CPU utilization and the average number of clients for cluster of servers as well as the number of virtual servers in the cluster. What can be seen in the graphs is that when the response time goes outside its thresholds the autonomic manager calculates the number of servers to add. It initially adds another VM server - which decreases the number of clients making requests to the server and thus also the response time. Later, when the response time threshold is again breached it adds another two VM servers to the pool. In order to force the servers' response time to go outside the threshold, the number of clients is increased suddenly by the workload generator, as can be seen around 500s and 1700s.

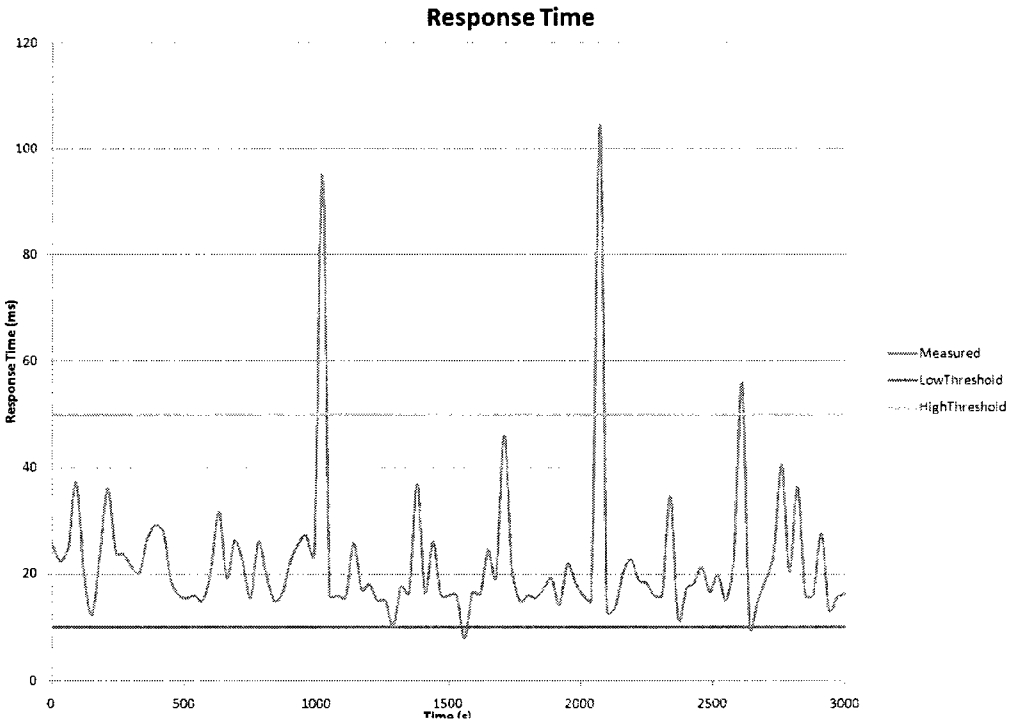

FIGURE 6.24: Response Time - Virtual Machines

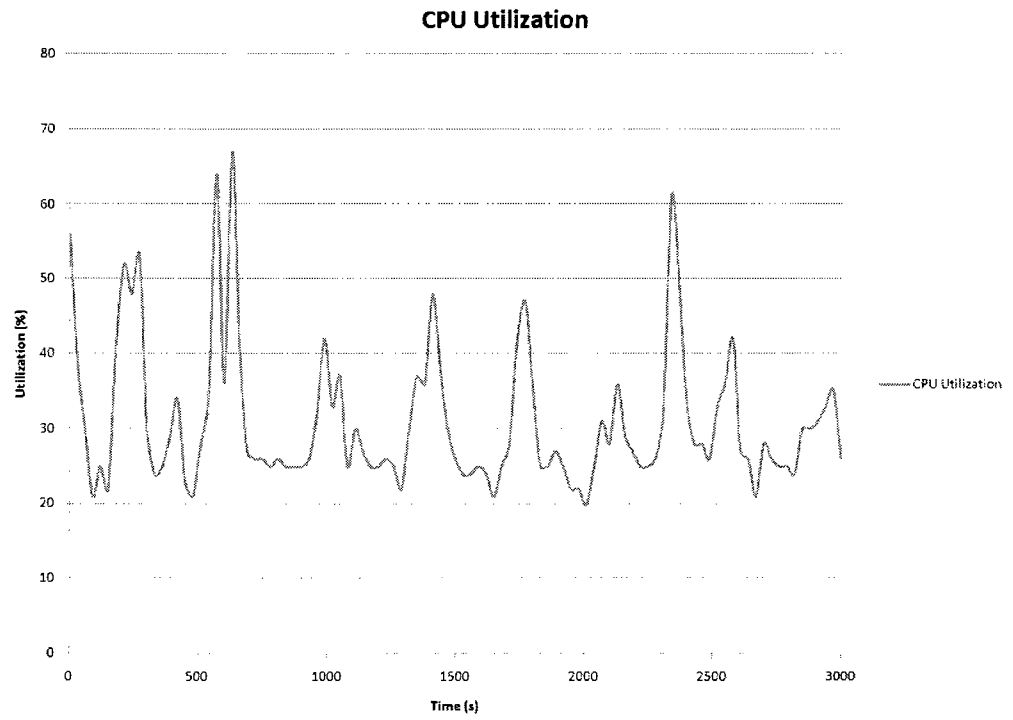

FIGURE 6.25: CPU Utilization - Virtual Machines

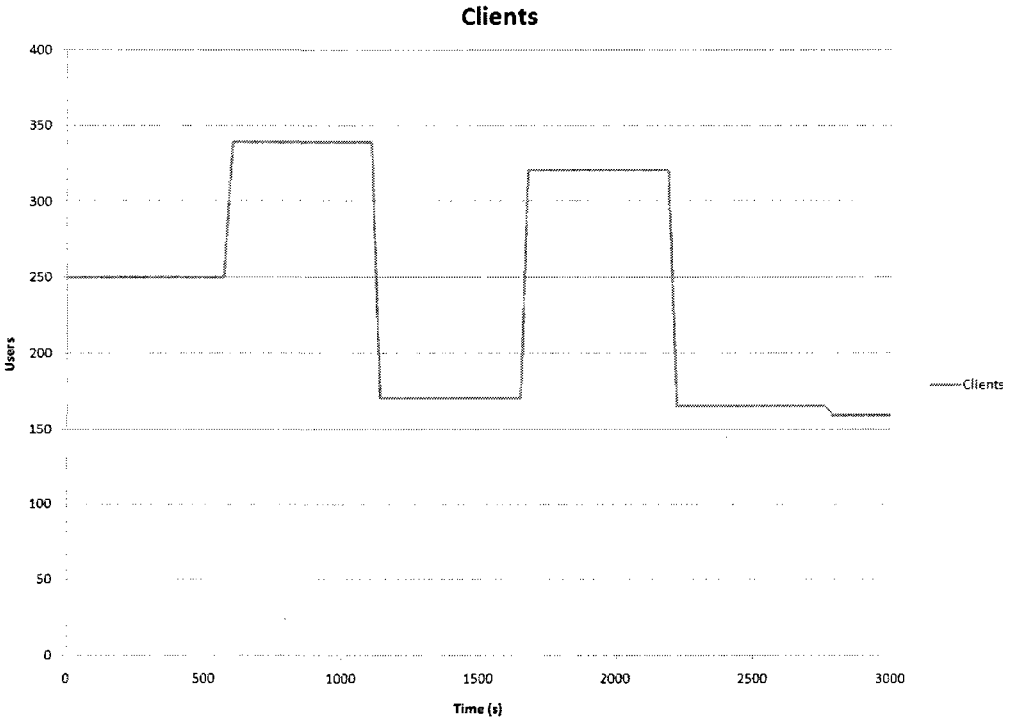

FIGURE 6.26: Clients - Virtual Machines

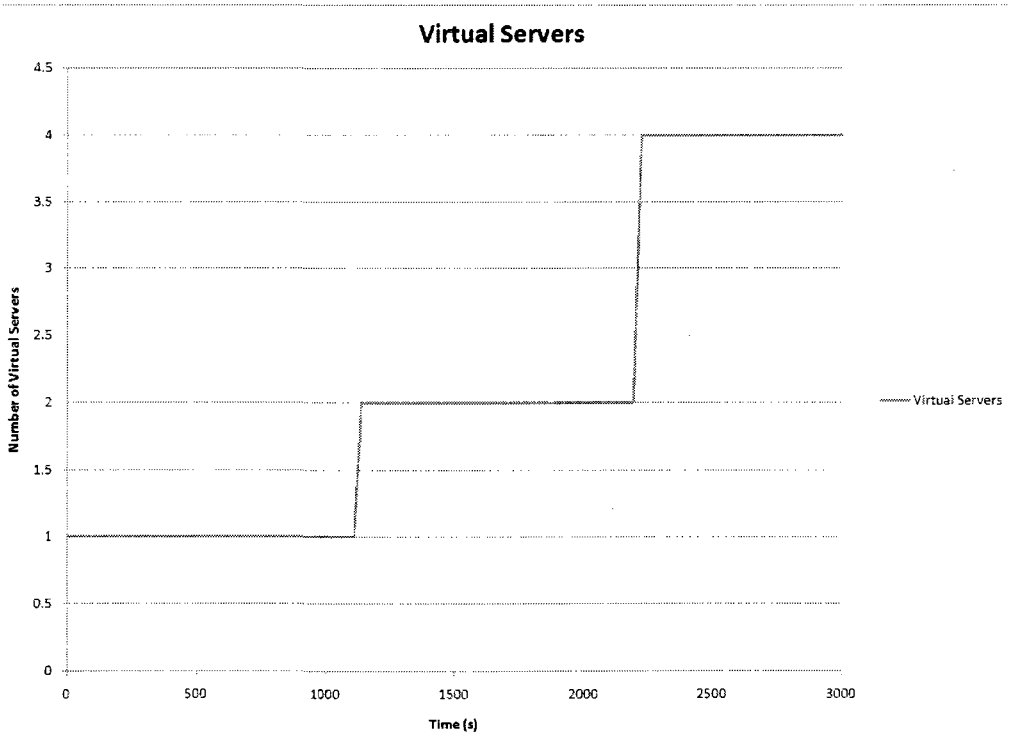

FIGURE 6.27: Servers - Virtual Machines

Chapter 7

Conclusions and Future Research

Since the launch of the Autonomic Computing Manifesto [4], autonomic computing has seen increased interest in both research and industry. In the future autonomic solutions will play an important role in managing data centers and helping system administrators quickly diagnose and repair problems in the resources that they manage. While autonomic computing is showing great promises, the lack of a standardized architecture can slow or even stop the adoption of autonomic solutions in data centers. Similarly, the lack of an architecture slows research in the area of autonomic computing as different research groups repeat the same work. The research presented in this thesis aims to solve this problem by creating a standard architecture and framework for autonomic systems.

7.1 Concepts Addressed in this Thesis

This thesis presents an architecture for autonomic system rooted in software engineering and real-time principles. The architecture splits the autonomic system into its composing parts and allows the creation of a distributed autonomic system where different components reside on different machines. The development of the architecture is done using the Model Driven Development approach. Furthermore, the architecture infuses real-time software patterns and mechanisms in order to ensure the reliability, fault tolerance and message synchronization of the created system. While the concept of a standardized architecture for autonomic systems

is not new in literature, the novelty of this thesis comes in the form of the distributed nature of the resulting architecture and in the use of real-time patterns to guarantee the correct execution of the system.

The thesis also presents a prototype framework built on top of the architecture. The framework takes the architecture and implements it using open standards. The specifications for each of the components are presented, as well as the communication, persistence and message validation subsystems. The framework allows autonomic computing component developers to concentrate on the specific functionality of the component and not on extraneous issues like data measurements, communications, system integrity, etc. Two test examples are presented which show how components can be reused for different autonomic systems which have different goals. Results of how the control loops behave are presented.

7.2 Contributions of this Thesis

This thesis examines the creation of an autonomic system architecture which can be standardized and used for all the aspects of autonomic computing represented by self-CHOP. An architecture rooted in real-time software patterns is presented and developed and extended into an autonomic system framework.

More specifically, the need for a standardized architecture and issues with which such an architecture must deal are presented. This includes a review of research done in autonomic system architectures and components, as well as a review of existing autonomic systems and standards that deal with aspects of autonomic computing. A full distributed architecture which makes use of real-time software patterns is created in order to address these issues. Mechanisms that ensure the correct execution of the autonomic computing control loop are presented and added to the architecture. On top of the architecture, a framework is built which uses the WSDM open standard in order to allow the components to communicate with each other. The design also includes issues of persistence for the state of components as well as the need for a registry which stores available components. Both the architecture and the framework are presented using the Unified Modeling Language. Two test systems are built on top of the framework and results obtained with the two systems are presented. The two systems demonstrate that it is possible to reuse components between different autonomic computing systems.

This thesis made a number of significant research contributions which can be summarized as follows:

1. A distributed architecture for autonomic computing systems which is rooted in real-time software patterns was presented. Through the creation of a standard architecture, which splits the system into eight basic component, each with very specific roles, the time to develop new autonomic solutions is decreased as researchers can concentrate on a small subset of the autonomic system.

2. The architecture provides synchronization, reliability and fault tolerance capabilities to the autonomic control loop. The architecture, due to its inovative distributed nature, must deal with issues of reliability and fault tolerance in its components, as well as message synchronization between components. The solutions to these problems are taken and adapted to the control loop from real time systems, which must always deal with these issues.

3. A new framework which implements the architecture is developed. The framework is developed on top of the WSDM open standard and makes use directly of a number of WSDM capabilities, while extending other capabilities. By using WSDM, the requirement that autonomic systems be developed on top of open standards, in order to allow interoperation between autonomic systems built by different vendors is achieved. The framework presented in this thesis is the first framework which achieves this very important reauirement of autonomic systems.

4. A new registry that stores existing components and autonomic systems is also developed. Like the requirement for use of open standards, an autonomic registry which provides capabilities for autonomic system components similar to what an UDDI registry provides for web services, is paramount for the adoption of autonomic systems. This thesis describes such a registry.

5. A GUI which allows system administrators to create an autonomic computing system by simply drag and dropping the various components on the interface. A third very important requirement for autonomic systems is that of ease of use. The GUI developed in this thesis supports the architecture and framework by allowing the creation of autonomic systems from components to be done in a very intuitive and easy manner.

6. A new model for application server response time calculations is developed through the analysis of the scheduling and queuing mechanisms taking place inside an application server. The non-linear, space-state set of equations discovered via this analysis are then used in an Extended Kalman Filter estimator to predict the future state of the application server. Finally, the model's parameters and sampling interval are identified through autocorrelatiuon and RPE approach.

7.3 Future Research

While this thesis presents a framework that would allow the composition of autonomic components in order to create an autonomic system that manages some resources, the implementation of a large number of these components is outside the scope of this thesis. As such, more components need to be developed in order to validate the architecture and ensure that the architecture can be applied to other types of components, like machine learning for example.

Another avenue for research is that of adaptation in the control loop itself. While this thesis only uses a simple adaptor that modifies the rest of the control loop, the adaptor itself can become a more complex control loop which manages the autonomic control loop, as presented in [9]. The research presented in [9] can be extended in order to analyze the impact of making changes in the control loop at runtime.

The model developed in this paper for the response time of an application server can also be improved, by applying identification methods to discover the initial parameters of the model's matrices. Various identification methods can be used and compared to see which gives the best results.

With the framework developed, future research can concentrate on developing various types of control loops. It would be useful if a baseline for comparison of control performance would be created, similar to the UCI Machine Learning [56] repository which exists for machine learning algorithms.

Bibliography

[1] M. Doser. (1997, March) Black monday: The stock market crash of 1987. [Accessed: May 2009]. [Online]. Available: http://www.newlearner.com/courses/hts/bec4a/ecoho16.htm

[2] H. J. Foxwell. (2007, March) Virtualization: What's old is new again. Sun Microsystems. [Accessed: May 2009]. [Online]. Available: http://regions.cmg.org/regions/ncacmg/downloads/mar082007_session1.pdf

[3] Optimizing data centers for high-density computing. HP technology brief. Hewlett Packard. [Accessed: May 2009]. [Online]. Available: http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00064724/c00064724.pdf

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

[5] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu, "An autonomic computing approach to server virtualization," in *I2TS '07: Proceedings of the 2007 International Information and Telecommunication Technologies Symposium*, 2007, pp. 222–236.

[6] 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.

[7] D. Ionescu, B. Solomon, M. Litoiu, and M. Mihaescu, "Web service distributed management framework for autonomic server virtualization," in *SVM '08: Proceeding of the 2008 Systems and Virtualization Management. Standards and New Technologies.* Springer Berlin Heidelberg, 2008, pp. 61–71.

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

[9] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu, "A real-time adaptive control of autonomic computing environments," in *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research.* New York, NY, USA: ACM, 2007, pp. 124–136.

[10] D. Ionescu, B. Solomon, M. Litoiu, and M. Mihaescu, "A robust autonomic computing architecture for server virtualization," in *International Conference on Intelligent Engineering Systems, 2008. INES 2008.*, Feb. 2008, pp. 173–180.

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

[12] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu, "Decentralized predictive control of autonomic computing environments," in *I2TS'2006: Proceedings of the 2006 International Information and Telecommunication Technologies Symposium*, 2006, pp. 94–103.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] G. Welch and G. Bishop, "An introduction to the Kalman Filter," University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, Tech. Rep., July 2006, [Accessed: May 2009]. [Online]. Available: http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf

[26] W. H. F. Aly and H. Lutfiyya, "Feedback control theoretic technique for data centers," in *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems.* Washington, DC, USA: IEEE Computer Society, 2007, p. 38.

[27] M. Salehie and L. Tahvildari, "A policy-based decision making approach for orchestrating autonomic elements," in *STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 173–181.

[28] B. Simmons and H. Lutfiyya, "Policies, grids and autonomic computing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[29] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos, "Requirements-driven design of autonomic application software," in *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research.* New York, NY, USA: ACM, 2006, p. 7.

[30] E. Kasten and P. McKinley, "MESO: Supporting online decision making in autonomic computing systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, no. 4, pp. 485–499, April 2007.

[31] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos, "A control theoretical approach to self-optimizing block transfer in web service grids," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 2, pp. 1–30, 2008.

[32] IBM. IBM - IBM Tivoli Intelligent Orchestrator - Tivoli Intelligent Orchestrator - Software. IBM. [Accessed: May 2009]. [Online]. Available: http://www-01.ibm.com/software/tivoli/products/intell-orch/

[33] Sun N1 Service Provisioning System. SUN. [Accessed: May 2009]. [Online]. Available: http://www.sun.com/software/products/service_provisioning/index.xml

[34] Enigmatec. Enigmatec Virtual Orchestrator. Enigmatec. [Accessed: May 2009]. [Online]. Available: http://www.enigmatec.com/products/enigmatec_virtual_orchestrator

[35] Intridea, "Scalr," Intridea, [Accessed: May 2009]. [Online]. Available: https://scalr.net/

[36] *ARM*, Open Group Std., [Accessed: May 2009]. [Online]. Available: http://www.opengroup.org/tech/management/arm/

[37] *Web Service Distributed Management*, OASIS Std., [Accessed: May 2009]. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm

[38] *Web Services for Management*, DMTF Std., [Accessed: May 2009]. [Online]. Available: http://www.dmtf.org/standards/wsman/

[39] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[40] A. Burns, *Real-Time Systems and Programming Languages.* Boston: Addison-Wesley, 1996.

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

[42] L. BenMohamed and S. Meerkov, "Feedback control of congestion in store-and-forward datagram networks: the case of a single congested node," in *Decision and Control, 1992., Proceedings of the 31st IEEE Conference on*, vol. 1, 1992, pp. 991–996.

[43] L. Ljung, *Theory and Practice of Recursive Identification.* Cambridge: MIT Press, 1983.

[44] W. W. Zhou and M. Blanke, "Identification of a class of nonlinear state-space models using RPE techniques," in *Decision and Control, 1986 25th IEEE Conference on*, vol. 25, Dec. 1986, pp. 1637–1642.

[45] A. Gorji and M. Menhaj, "Identification of nonlinear state space models using an mlp network trained by the em algorithm," in *Neural Networks, 2008.*

IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on, June 2008, pp. 53–60.

[46] B. P. Douglass, *Real-Time Design Patterns.* Addison-Wesley, 2003.

[47] "Apache muse," Apache, [Accessed: May 2009]. [Online]. Available: http://ws.apache.org/muse/

[48] "JBoss Application Server," JBoss, [Accessed: May 2009]. [Online]. Available: http://www.jboss.org/

[49] "WebSphere Application Server," IBM, [Accessed: May 2009]. [Online]. Available: http://www-01.ibm.com/software/webservers/appserv/was/

[50] H. Kreger, "A little wisdom about WSDM," IBM, March 2005, [Accessed: May 2009]. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-wisdom/

[51] "Enterprise JavaBeans 3 specification," Sun Microsystems, [Accessed: May 2009]. [Online]. Available: http://java.sun.com/products/ejb/docs.html

[52] M. Woodside, T. Zheng, and M. Litoiu, "Service system resource management based on a tracked layered performance model," in *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 175–184.

[53] "Application Performance Evaluator and Resource Allocation Tool: Overview," IBM, May 2003, [Accessed: May 2009]. [Online]. Available: http://www.alphaworks.ibm.com/tech/apera

[54] "Xen hypervisor," Xen, [Accessed: May 2009]. [Online]. Available: http://www.xen.org/

[55] D. A. Menasce and M. N. Bennani, "Autonomic virtualized environments," in *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems.* Washington, DC, USA: IEEE Computer Society, 2006, p. 28.

[56] A. Asuncion and D. Newman, "UCI machine learning repository," 2007, [Accessed: May 2009]. [Online]. Available: http://www.ics.uci.edu/~mlearn/MLRepository.html