

American University in Cairo

AUC Knowledge Fountain

Archived Theses and Dissertations

Resource optimization property manager for autonomic computing

Hazem A. M. Sharaf el Din

Follow this and additional works at: https://fount.aucegypt.edu/retro_etds



Part of the [Software Engineering Commons](#)

2004/12

The American University in Cairo
School of Science and Engineering

Resource Optimization Property Manager For Autonomic Computing

A Thesis Submitted to

Computer Science Department

in partial fulfillment of the requirement for

the degree of Master of Science

by

Hazem Ahmed Mohamed Sharaf El Din

Bachelor of Science, Computer Science

Under the supervision of

Dr. Hoda Hosny

And

Co-Supervision of

Dr. Amir Zeid

April 2004

2004/12

The American University in Cairo

Resource Optimization Property Manager For Autonomic Computing


A Thesis Submitted by

Hazem Ahmed Mohamed Sharaf El Din
to the
Department of Computer Science
May 2004

in partial fulfillment of the requirements for
The degree of Master of Science

has been Approved by :

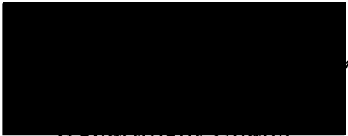
Dr. Hoda M. Hosny
Thesis Committee Chair/Advisor 
Affiliation : Department of Computer Science, The American University in Cairo

Dr. Amir Zeid
Thesis Committee Chair/Co-Advisor 
Affiliation : Department of Computer Science, The American University in Cairo

Dr. Gamal M. Aly
Thesis Committee Reader/Examiner 
Affiliation: Computer and Systems Engineering Department, Ain Shams University

Dr. Ahmed Sameh Mohamed
Thesis Committee Reader/Examiner 
Affiliation : Department of Computer Science, The American University in Cairo

Dr. Sherif El Kassas
Thesis Committee Reader/Examiner 
Affiliation : Department of Computer Science, The American University in Cairo


Program Chair

27/5/04
Date


Dean

July 31, 2004
Date

Dedication

To My Family

To My Doctors

To My Friends

Acknowledgment

Every time I was reading a thesis work of any person, I was pulled towards reading the acknowledgment section wishing to reach the time in which I will start writing my own acknowledgment section and finally here I'm.

First of all thanks to God that I was able to accomplish this research work successfully. I really cannot express my appreciation to my parents and sister for encouraging me and supporting me during the whole time of graduate studies.

I would like to thank my supervisor Dr. Hoda Hosny for her great effort and time in supporting me to accomplish my thesis work. I really believe that Dr. Hoda is one of the unique persons and professors that I will ever meet in my life. She is a very dedicated person towards her work and students all the time. I really feel that I'm very lucky to have her as my supervisor. I also would like to thank Dr. Amir Zeid for his co-supervision and feedback regarding my thesis work.

I would also like to thank the committee members Dr. Ahmed Sameh and Dr. Shereif El-Kassas for their efforts in reviewing and evaluating my research work. I would also like to thank the external examiner Dr. Gamal El-Din Aly for his great effort and useful comments regarding my research work. Finally I would like to thank the Computer Science Department for their support and efforts during my graduate studies period.

Special thanks goes to all my friends who always encouraged and supported me in achieving this work.

Abstract

Autonomic Computing is emerging as a significant new approach for the design of computing systems. Its goal is the production of systems that are self-managing, self-healing, self-protecting and self-optimizing. The high-tech industry has spent decades creating computer systems with ever-mounting degrees of complexity to solve a wide variety of business problems. Ironically, complexity itself has become part of the problem. This is actually the core of the autonomic computing dilemma. The question is how to provide such a promising system with the least possible level of complexity in order to avoid going into the same circle of infinitely cascaded complexity? Another important issue is that the fate of all the multi billion software products and modules that are already produced by large software companies or being used by large businesses in the IT-industry and even non IT-industry, is going to be determined by the planned design direction that is going to be pursued in the autonomic computing industry. One important question is whether the intended design direction is going to easily integrate the already developed and existing software products and solutions into the upcoming autonomization revolution or not!

This research work realizes the autonomic computing complexity from a different dimension, which does not completely solve the previously mentioned dilemmas, but it proposes a solution that might lead to a flexible approach for dealing with the problem. As the main goal of autonomic computing is to deliver a system that is capable of self-management, it actually means that each autonomic system should have the capabilities, skills, and experience to maintain each of those properties appropriately. Most probably such a system will implicitly denote a large and complex set of subsystems and this is what leads to the complexity of the provided solutions. In this research effort we propose a new notion to the autonomic computing architecture known as the property manager. The autonomic property manager is an autonomic manager that is capable of maintaining the management of any of the autonomic computing properties. For each property of the autonomic system an autonomic property manager will be dedicated to handle the duties of this property, i.e. a property keeper. In this research work we also present a brief description

for our proposed design of one autonomic property manager, namely the resource optimization property manager, which is responsible for managing the resource optimization of all the modules registered under its domain. By setting a specialization for each autonomic manager we can expect better performance and details abstraction. This research work also proposes some solutions to other critical issues that the autonomic systems will have to handle such as: the ability to provide goal specification and policies at the system management level for each property and to support high level goals at the business level. By providing a hierarchy of policy definitions at the system level that is controlled by a global system policy for each group of systems, we are able to reach a high-level policy definition that matches the business goals.

We conducted five practical experiments, which include some scenarios that test the validity of the proposed design for the autonomic property manager using the prototype implementation of the resource optimization property manager. The experiments demonstrate the ability of the resource optimization property manager to make decisions based on the predefined policies that contribute to the enhancement of the performance of the high-level system definition. The definitions were included in one of the policy files to specify the desired response time for the website used in our experiments. They also demonstrated the facilities that the resource optimization property manager can provide to the other registered systems by using the resource allocation mechanism to get more resources whenever possible in addition to the other services it provides. Both the architecture chapter and the experimental work chapter include the details of the used architecture and the conducted experiments throughout this research work, which finally proves the effectiveness of our proposed approach.

TABLE OF CONTENTS

Chapter 1. Introduction.....	1
1.1 The Autonomic Era.....	2
1.2 Problem Definition	3
1.3 Research motivation and objective.....	3
1.4 Research results.....	4
1.5 Document Organization.....	6
Chapter 2. Background And Related Work	7
2.1 Background.....	8
2.2 Related Work.....	14
2.2.1 Auto Tune Agents.....	14
2.2.2 Autonomia.....	18
2.2.3 An architecture for Autonomic Personal Computing Systems.....	20
2.2.3.1 Directions toward an autonomic framework	22
2.2.3.2 Autonomic managers	23
2.2.4 An architectural blueprint for autonomic computing.....	24
2.2.5 E-Model Framework.....	30
2.2.6 AutoMate.....	34
2.2.7 Business Workload Manager Prototype (BWLM).....	38
2.2.7.1 eWorkload Management overview	39
2.2.7.2 Management environment	40
2.2.8 Summary and Discussion of Related Work.....	41
Chapter 3. Proposed Architecture for the Autonomic Property Manager	45
3.1 Introduction.....	46
3.2 Autonomic Levels.....	46
3.3 Architecture Essence.....	50
3.3.1 Top Down Architecture Description.....	50
3.3.2 GUI Description.....	50
3.3.3 Sensors (Actuators) and Effectors.....	56
3.3.4 Policy Files.....	57
3.3.5 Knowledge Base.....	59
3.3.6 Extension Policy File.....	62
3.3.7 Event Notification.....	67
3.3.7.1 Event subscription	67
3.3.7.2 Delivery Chain example	68
3.3.7.3 Priorities	68
3.3.7.4 Missing features	69
3.3.8 Messaging Using JMS.....	70
3.4 The Server Side.....	74
3.4.1 Command Dispatcher.....	79
3.4.2 Resource Allocation.....	79
3.5 The Client Side.....	83

LIST OF FIGURES

FIGURE 2.1: ARCHITECTURE OF THE BAS AUTOTUNE AGENT [41].....	16
FIGURE 2.2: AUTONOMIA SYSTEM ARCHITECTURE [36].....	19
FIGURE 2.3: THE ARCHITECTURE OF AN AUTONOMIC ELEMENT [11].....	21
FIGURE 2.4: AN EXAMPLE OF TWO HIERARCHIES OF AUTONOMIC CONTROL [11].....	22
FIGURE 2.5: INTELLIGENT CONTROL LOOPS IN AN AUTONOMIC SYSTEM [1].....	25
FIGURE 2.6: THE FUNCTIONAL DETAILS FOR AN AUTONOMIC MANAGER [1].....	27
FIGURE 2.7: THE EMODEL XML-BASE MODEL BUILDING PROCESS [10].....	33
FIGURE 2.8: AUTOMATE ARCHITECTURE DIAGRAM [31].....	36
FIGURE 2.9: EWLM SAMPLE ENVIRONMENT [9].....	39
FIGURE 3.1: LEVELS OF AUTONOMIC REPRESENTATION.....	49
FIGURE 3.2: AUTONOMIC PROPERTY MANAGER USE CASE.....	51
FIGURE 3.3: ADD NEW SYSTEM GUI SNAPSHOT.....	53
FIGURE 3.4: ADD NEW SUBSYSTEM GUI SNAPSHOT.....	53
FIGURE 3.5: SYSTEM PEAK TIMES DEFINITION GUI SNAPSHOT.....	53
FIGURE 3.6: DB ER DIAGRAM.....	55
FIGURE 3.7: ADMIN CONSOLE SNAPSHOT.....	55
FIGURE 3.8: WEB POLICY PRECONDITION SECTION.....	61
FIGURE 3.9: WEB POLICY ACTION SECTION.....	61
FIGURE 3.10: EFFECTOR FUNCTION CALL SNAPSHOT.....	66
FIGURE 3.11: EFFECTOR POLICY CALL SNAPSHOT.....	66
FIGURE 3.12: EVENT DELIVERY CHAINS [14].....	68
FIGURE 3.13: PTP MESSAGING [31].....	71
FIGURE 3.14: PROPERTY MANAGER OVERALL ARCHITECTURE.....	73
FIGURE 3.15: PROPERTY MANAGER CLASS DIAGRAM.....	77
FIGURE 3.16: PROPERTY MANAGER OVERALL ARCHITECTURE.....	78
FIGURE 3.17: COMMAND DISPATCHER ACTIVITY DIAGRAM.....	80
FIGURE 3.18: OFF PEAK TIME VIEW.....	82
FIGURE 3.19: RESOURCE ALLOCATION ACTIVITY DIAGRAM.....	82
FIGURE 3.20: COMMAND RECEIVER ACTIVITY DIAGRAM.....	83
FIGURE 4.1: APACHE SERVER TEST 1.....	92
FIGURE 4.2: ACTIVE USERS VS TEST ELAPSED TIME.....	93
FIGURE 4.3: RESPONSE TIME VS TEST ELAPSED TIME.....	93
FIGURE 4.4: HTTP ERRORS VS CONCURRENT HTTP.....	94
FIGURE 4.5: HTTP ERRORS VS ELAPSED TIME.....	94
FIGURE 4.6: TEST ENVIRONMENT SETUP.....	95
FIGURE 4.7: HTTP ACTIVE USERS VS ELAPSED TIME.....	95
FIGURE 4.8: HTTP RESPONSE TIME VS ELAPSED TIME.....	96
FIGURE 4.9: HTTP ERRORS VS CONCURRENT HTTP REQUESTS.....	97
FIGURE 4.10: HTTP ERRORS VS ELAPSED TIME.....	98
FIGURE 4.11: TEST THREE ENVIRONMENT SETUP.....	99
FIGURE 4.12: HTTP RESPONSE TIME VS ELAPSED TIME.....	100

FIGURE 4.13: HTTP ERRORS VS CONCURRENT HTTP REQUESTS	101
FIGURE 4.14: WEB POLICY PRECONDITION SECTION	104
FIGURE 4.15: WEB POLICY ACTION SECTION.....	104
FIGURE 4.16: HTTP POLICY PRECONDITION SECTION	105
FIGURE 4.17: HTTP POLICY ACTION SECTION.....	105
FIGURE 4.18: TEST 1 ENVIRONMENT SETUP.....	106
FIGURE 4.19: HTTP ACTIVE USERS VS ELAPSED TIME.....	107
FIGURE 4.20: HTTP RESPONSE TIME VS ELAPSED TIME.....	108
FIGURE 4.21: EXPERIMENT THREE ENVIRONMENT SETUP.....	110
FIGURE 4.22: POLICY PRECONDITION SECTION	113
FIGURE 4.23: POLICY ACTION SECTION.....	113
FIGURE 4.24: HTTP ACTIVE USERS VS ELAPSED TIME.....	115
FIGURE 4.25: HTTP RESPONSE TIME VS ELAPSED TIME.....	115
FIGURE 4.26: RESPONSE TIME VS NUMBER OF RESPONSES.....	116
FIGURE 4.27: HTTP ERRORS VS ELAPSED TIME	116
FIGURE 4.28: SECURITY POLICY PRECONDITION SECTION	119
FIGURE 4.29: SECURITY POLICY ACTION SECTION	119
FIGURE 4. 30: EXPERIMENT FOUR TEST ENVIRONMENT SETUP	121
FIGURE 4. 31: RESPONSE TIME FOR TEST CASE 1	123
FIGURE 4. 32: RESPONSE TIME FOR TEST CASE 2	123
FIGURE 4. 33: APPLICATION LOG FILE.....	125
FIGURE 4. 34: NUMBER OF CALLS MADE PER POLICY	126
FIGURE 4.35: NUMBER OF CALLS MADE PER POLICY IN TEST 2	127
FIGURE 4.36: SENSOR READINGS DURING A TEST ELAPSED TIME	128

Chapter 1. Introduction

1.1 The Autonomic Era.

Computing systems will soon become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. When they do reach such a level of complexity, there will be no way to make timely decisive response to the rapid stream of changing and conflicting demands. Also the need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces a new level of complexity [28]. The most lucrative alternative at such a time will be the adoption of autonomic computing. The term autonomic is derived from human biology. The autonomic nervous system monitors our heartbeats, checks our blood sugar level and keeps our body temperature close to 98.6 °F, without any conscious effort on your part. In much the same way, autonomic computing components anticipate computer system needs and resolve problems with minimal human intervention [1]. Autonomic computing systems will be able to manage themselves given high-level objectives from administrators. Thus it will alleviate the administration complexities from the shoulders of the system administrators and will introduce more robust and efficient systems to the business. At present there is no single specific technology known as autonomic computing but in general any autonomic system should be able to maintain Self-management, which means a system that functions well without our regular interference to provide a simplified user experience. That system should possess the following features or characteristics:

- **Self Configuration**
- **Self Optimization**
- **Self Healing**
- **Self Protection**

Only recently IBM has introduced major initiatives in leading the autonomic computing research towards formalizing a clear vision of the autonomic computing design and framework by defining the major properties of an autonomic system. Many researchers have contributed to the autonomic computing field by introducing proposed designs and frameworks for the autonomic computing in an effort to reach a completely self-managed

autonomic system. However, it seems that the proposed models and frameworks are still faraway from forming a perfect and a complete autonomic system.

1.2 Problem Definition

The high-tech industry has spent decades creating computer systems with ever-mounting degrees of complexity to solve a wide variety of business problems. Ironically, complexity itself has become part of the problem [1]. This is actually the core of the autonomic computing dilemma. The question is how to provide such a promising system with the least possible level of complexity in order to avoid going into the same circle of infinitely cascaded complexity? Another important issue is that the fate of all the multi billion software products and modules that are already produced by large software companies or being used by large businesses in the IT-industry and even non IT-industry, is going to be determined by the planned design direction that is going to be pursued in the autonomic computing industry. One important question is whether the intended design direction is going to easily integrate the already developed and existing software products and solutions into the upcoming autonomization revolution or not! On one hand most of the conducted research work in the field of autonomic computing is either focusing on the software models and patterns that can be used to produce an autonomic system from scratch or it provides an autonomic model which only fits one property of the autonomic system properties. On the other hand most of the tools that are produced by some of the companies to help in accomplishing an autonomic system are software specific products, which are not intended to cover other competitive products. All these factors together develop the need for additional research work in this area in order to provide new ideas and models that can serve the autonomic field.

1.3 Research motivation and objective

This research work realizes the autonomic computing complexity from a different dimension, which does not completely solve the previously mentioned dilemmas, but it proposes a solution that might lead to a flexible approach for dealing with the problem. As the main goal of autonomic computing is to deliver a system that is capable of self-managing, self-healing, self-protecting and self-optimizing, it actually means that each autonomic

system should have the capabilities, skills, and experience to maintain each of those properties appropriately. Most probably such a system will implicitly denote a large and complex set of subsystems and this is what leads to the complexity of the provided solutions. In this research effort we propose a new notion to the autonomic computing architecture known as the property manager. The aim of the property management concept is to represent each autonomic property separately by an autonomic property manager, which is capable of maintaining and handling the duties of the property it represents. The net result is that we can decompose the embedded complexity of any autonomic system and reach a more powerful system that is easier to maintain and run. This research work also propose some solutions to other critical issues that the autonomic systems will have to handle such as: the ability to provide goal specification and policies at the system management level for each property and to support high level goals at the business level. By providing a hierarchy of policy definitions at the system level that is controlled by a global system policy for each group of systems, we are able to reach a high-level policy definition that matches the business goals. Finally, the real motivation behind this research is to join the ongoing efforts and contribute to the autonomic computing era by introducing the new autonomic architecture of the autonomic property manager to the autonomic computing field.

1.4 Research results

In our proposed design we were seeking an approach that would simplify some of the complexities imposed by the nature of the autonomic computing system. The notion of the property manager was seen to guarantee the knowledge and professionalism of specialization. What this means is that each property manager will be responsible and capable of maintaining one of the autonomic properties appropriately by enclosing the required knowledge and tools to do so. This way a group of property managers, each maintaining its respective role, can cooperate together to form an autonomic system, which is capable of maintaining the set of autonomic properties together. This overall setup would finally lead to a self management-capable system. In this research work we propose a flexible architecture for the autonomic property manager, which offers one possible setup that satisfies most of the required features to compose a self-management capable system.

Since it is very difficult (in terms of the time frame and available resources) to design and implement each of the four basic autonomic property managers which together form the Self Management autonomic property (Configuration Manager, Self Healing Manager, Security Manager, Optimization Manger), we implemented a basic prototype for the resource optimization property manager as a proof of concept for the efficiency of our proposed model. Our resource optimization manager was prototyped using an architectural model, which can just as well be used by any of the property managers. The resource optimization manager deals mainly with two subsystems, which fit under its management: A web server and an application server. Together they form the abstract level of a website resource optimization management team. In general a policy definition is provided for each registered subsystem. As for our prototype a policy definition is provided for the HTTP server, and another one for the application server. The same could be applied to any other subsystem which is part of the high level system (i.e. the website in our case) or any system, which the resource optimization manager will be managing. This way the resource optimization manager can dig out through the different layers from which the main system is composed and abstract those details at the very first level in which the business goals are defined. /in the case of the website, the highest level of goal definition could include the definition of the desired average response time of the website. The required actions to be taken upon the violation of the defined goal, are included in the policy files. We conducted five practical experiments, which include some scenarios that test the validity of most of the above concepts about the autonomic property manager notion. Since our main focus is on the resource optimization manager as a proof of concept, our experiments were only dealing with the implemented prototype of the resource optimization property manager.

The experiments demonstrate the ability of the resource optimization manager to make decisions based on the predefined policies that contribute to the enhancement of the performance of the high-level system definition. The definitions were included in one of the policy files to specify the desired response time for the website which was used in our experiments. They also demonstrated the facilities that it can provide to the other registered systems by using the resource allocation mechanism to get more resources whenever possible in addition to the other services provided the property manager. Both the architecture chapter

and the experimental work chapter include the details of the used architecture and the conducted experiments.

1.5 Document Organization.

The organization of this research work is as follows. Chapter one is an introductory chapter, in which we present a brief overview of the autonomic computing systems and provide a summary for the autonomic problem definition. Then we mentioned the motivation behind our research and finally we presented a brief description for the research results. Chapter two contains a brief description concerning the autonomic computing background and a survey of the most recent and related work that we referred to during our work. Chapter three presents the problem that this research work addresses and introduces the proposed solution to the addressed problem. In this chapter we provide a detailed description of our proposed design model and used technologies that were used in our prototype implementation and experimental work. Chapter Four presents the experimental work and analysis of results of work in details. Finally chapter five concludes the thesis work, and discusses the current limitations and directions for future work.

Chapter 2. Background And Related Work

2.1 Background.

Imagine if you could describe the business functions that you want your system to provide and it just took care of itself! For example all your needed software would be located, installed, and configured automatically. Resources would become available when they are needed and are freed when they aren't [12]. As networks and distributed systems grow and change, they can become increasingly hampered by system deployment failures, hardware and software issues, not to mention human error [1]. Such scenarios in turn require further human intervention to enhance the performance and capacity of IT components. This drives up the overall IT costs even though the technology component costs continue to decline. As a result, many IT professionals seek ways to improve their return on investment(ROI)* in their IT infrastructure, by reducing the total cost of ownership (TCO)* of their environments while improving the quality of service (QoS)* for users. We do not see a slowdown in Moore 's law* as the main obstacle to further progress in the IT industry. Rather, it is our industry 's exploitation of the technologies that have arisen in the wake of Moore 's law that have led us to the verge of a complexity crisis [1]. Software developers have fully exploited a four-to-six order-of-magnitude increase in computational power by producing ever more sophisticated software applications and environments. There has been an exponential growth in the number and variety of systems and components in recent years. The value of database technology and the Internet has fueled significant growth in storage subsystems, which are now capable of holding petabytes of structured and unstructured information. Networks have interconnected our distributed, heterogeneous systems. Our information society presently creates unpredictable and highly variable workloads on those networked systems. And today, these increasingly valuable, complex systems require more and more skilled IT professionals to install, configure, operate, tune and maintain. Autonomic computing helps address these complexity issues by using technology to manage technology. The idea is not new, many of the major players in the industry have developed and delivered products based on this concept. The term autonomic is derived from human biology. The autonomic nervous system monitors your heartbeat, checks your blood sugar level and keeps your body temperature close to 98.6 °F, without any conscious effort on your part. In much the same way, autonomic computing components anticipate computer system needs and resolve

problems with minimal human intervention. However, there is an important distinction between autonomic activity in the human body and autonomic responses in computer systems. Many of the decisions made by autonomic elements in the body are involuntary, whereas autonomic elements in computer systems make decisions based on tasks you choose to delegate to the technology. In other words, adaptable policy rather than rigid hard coding determines the types of decisions and actions autonomic elements make in computer systems. Autonomic computing can result in a significant improvement in system management efficiency, when the disparate technologies that manage the environment work together to deliver performance results system-wide. For this to be possible in a multi-vendor infrastructure, however, IBM and other vendors must agree on a common approach to architecting autonomic systems [1][28][33].

Therefore, in mid-October 2001, IBM released a manifesto [33] observing that the main obstacle to further progress in the IT industry is a looming software complexity crisis. The company cited applications and environments that weigh in at tens of millions of lines of code and require skilled IT professionals to install, configure, tune, and maintain. The manifesto pointed out that the difficulty of managing today's computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity. Computing systems' complexity appears to be approaching the limits of human capability, yet the march toward increased interconnectivity and integration rushes ahead unabated [1].

This march could turn the dream of pervasive computing with trillions of computing devices connected to the Internet into a nightmare. Programming language innovations have extended the size and complexity of systems that architects can design. Relying solely on further innovations in programming methods will not get us through the present complexity crisis. As systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. Soon systems will become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. And there will be no

way to make timely, decisive responses to the rapid stream of changing and conflicting demands. Here comes the option of Autonomic computing [28].

The term autonomic computing is emblematic of a vast and somewhat tangled hierarchy of natural self-governing systems, many of which consist of a myriad of interacting, self-governing components which in turn comprise large numbers of interacting, autonomous, self-governing components at the next level down. The enormous range in scale, starting with molecular machines within cells and extending to human markets, societies, and the entire world socioeconomy, mirrors that of computing systems, which run from individual devices to the entire Internet. Thus, we believe it will be profitable to seek inspiration in the self-governance of social and economic systems as well as purely biological ones. Clearly then, autonomic computing is a grand challenge that reaches far beyond a single organization. Its realization will take a concerted, long-term, worldwide effort by researchers in a diversity of fields. The main properties that every autonomic system is expected to satisfy are as follow [7] [28][33]:

- **Self Management**

The essence of autonomic computing systems is selfmanagement, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7. In extreme cases, if a system is not able to satisfy the assigned policies for some reason, human intervention will be called upon.

- **Self Configuration**

Installing, configuring, and integrating large, complex systems is challenging, time-consuming, and error-prone even for experts. Most large Web sites and corporate data centers are haphazard accretions of servers, routers, databases, and other technologies on different platforms from different vendors. It can take teams of expert programmers months to merge two systems or to install a major e-commerce application such as SAP. Autonomic systems will configure themselves automatically in accordance with high-level policies representing business-level objectives, for example, that specify what is desired, not how it is

to be accomplished. When a component is introduced, it will incorporate itself seamlessly, and the rest of the system will adapt to its presence much like a new cell in the body or a new person in a population.

- **Self Optimization**

Complex middleware, such as Web Sphere, or database systems, such as Oracle or DB2, may have hundreds of tunable parameters that must be set correctly for the system to perform optimally, yet few people know how to tune them. Such systems are often integrated with other, equally complex systems. Consequently, performance tuning of one large subsystem can have unanticipated effects on the entire system. Autonomic systems will continually seek ways to improve their operation, identifying and seizing opportunities to make themselves more efficient in performance or cost.

- **Self Healing**

IBM and other IT vendors have large departments devoted to identifying, tracing, and determining the root cause of failures in complex computing systems. Serious customer problems can take teams of programmers several weeks to diagnose and fix, and sometimes the problem disappears mysteriously without any satisfactory diagnosis. Autonomic computing systems will be expected to detect, diagnose, and repair localized problems resulting from bugs or failures in software and hardware, perhaps through a regression tester. Using knowledge about the system configuration, a problem diagnosis component (based on a Bayesian network, for example) would analyze information from log files, possibly supplemented with data from additional monitors that it has requested. The system would then match the diagnosis against known software patches (or alert a human programmer if there are none), install the appropriate patch, and retest.

- **Self Protection**

Despite the existence of firewalls and intrusion-detection tools, humans must at present decide how to protect systems from malicious attacks and inadvertent cascading failures. Autonomic systems will be self-protecting in two senses. They will defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading

failures that remain uncorrected by self-healing measures. They will also anticipate problems based on early reports from sensors and take steps to avoid or mitigate them.

The autonomic computing vision is the analogous situation for IT. Autonomic computing allows people to focus on the big picture because the low level tasks can be "taught" to monitor and manage themselves [12]. Alan Ganek, IBM vice President of Autonomic Computing, explained in a session at developWorks live, that autonomic does not just mean automated. An automated system might simply specify that this server is assigned to a particular task between the hours of 4 and 7 and to a different task the remainder of the day. This specification may be correct and it may help to have it in place in your system. On the other hand, you may want a more business oriented rule being enforced. Your rule may be that gold-level customers can expect a response to be generated within two seconds while silver customers can expect a response to be generated within six seconds [12][33]. Although autonomic computing is not far away, Ganek recommends a step-by-step approach to evolve the infrastructure of any company in that direction. First you need to assess where you are in the continuum. Then you need to decide which area of complexity to tackle first. Ganek reminded the audience that "the complexity is at every level of the system. Autonomic is hardware, software, and system management." [12].

Ganek outlines five levels that run the gamut from manual to autonomic. He says that most organizations are at level 1. This basic level requires that the IT staff install, monitor, maintain, and replace each system element. The second level, the managed stage, can at present be implemented using many of the tools that IBM and other vendors provide. The tools help the IT staff analyze system components and use the results to decide which actions to take. Ganek says that many state-of-the-art customers are currently at this level. Each level replaces some area of human intervention and decision-making. The predictive level provides additional features, which are built on the monitoring tools added in the previous level. At this third level, the system can correlate measurements and make recommendations. The IT staff looks to approve the recommendations and take actions. This leads to faster and better decision making. At level four, the staff becomes less involved in viewing the recommendations and taking actions. This is the adaptive level and features the ability of the

technology to make more of the decisions automatically. Staff members spend most of their time setting the policies and managing the controls. In many ways the technology at the autonomic level (fifth level) is similar to that introduced at level four. The difference is that the IT services are now integrated with business rules. This is where you stop defining IT rules in terms of the components and tuning parameters. Now the policies are set in terms of business logic, and the IT staff focuses on tuning the system and the rules so they best support the company bottom line. As an example, if a Web site supports free content and subscriber-only content, then a rule might specify that resources should be allocated so that the user experience for subscribers is at a certain level even if that means degrading the experience for non-paying site visitors [12][19].

While autonomic systems will assume much of the burden of system operation and integration, it will still be up to humans to provide these systems with policies, the goals and constraints that govern their actions. The enormous leverage of autonomic systems will greatly reduce human errors, but it will also greatly magnify the consequences of any error humans do make in specifying goals. The indirect effect of policies on system configuration and behavior exacerbates the problem because tracing and correcting policy errors will be very difficult. It is thus critical to ensure that the specified goals represent what is really desired. Two engineering challenges stem from this mandate: ensure that goals are specified correctly in the first place, and ensure that systems behave reasonably even when they are not. In many cases, the set of goals to be specified will be complex, multidimensional, and conflicting. Even a goal as superficially simple as “maximize utility” will require a human to express a complicated multi-attribute utility function. A key to reducing error will be to simplify and clarify the means by which humans express their goals to computers. The second challenge ensuring reasonable system behavior in the face of erroneous input is another facet of robustness: Autonomic systems will need to protect themselves from input goals that are inconsistent, implausible, dangerous, or unrealizable with the resources at hand. Autonomic systems will subject such inputs to extra validation, and when self-protective measures fail, they will rely on deep-seated notions of what constitutes acceptable behavior to detect and correct problems. In some cases, such as resource overload, they will inform human operators about the nature of the problem and offer alternative solutions [28].

controller design agent that uses optimal control theory to derive a feedback control algorithm customized to that server, and a run-time control agent that deploys the feedback control algorithm in an on-line real-time environment to automatically manage the Web server. The designed autonomic feedback control system is able to handle the dynamic and interrelated dependencies between the tuning parameters and the performance metrics with guaranteed stability from control theory. The effectiveness of the AutoTune agents is demonstrated through experiments involving variations in workload, server capacity, and business objectives. The results also serve as a validation of the ABLE toolkit and the AutoTune agent framework.

The Agent Building and Learning Environment (ABLE) [24][25] is a Java based toolkit for developing and deploying hybrid intelligent agent applications. It provides a comprehensive library of intelligent reasoning and learning components packaged as Java beans (known as AbleBeans) and a lightweight Java agent framework to construct intelligent agents (known as Able-Agents). The AbleBean Java interface defines a set of common attributes (name, comment, state, etc.) and behaviors (standard processing methods such as `init()`, `reset()`, `process()`, and `quit()`), which allows AbleBeans to be connected to form AbleAgents. A Java Swing-based GUI, AbleEditor, is also provided for creating and configuring AbleBeans, and for constructing and testing the AbleAgents built from them. For most AbleBeans, the user interface is through a GUI component known as a “Customizer” that allows the user to set and view parameters related to the bean. The base AutoTune agent is a function-specific Able-Agent for autonomic computing. Inspired by human biology, the AutoTune agent is based on an architecture, which combines several elements that are useful in building systems to react to a dynamic environment. The AutoTune agent contains two basic building blocks (AbleBeans): the AutotuneController bean and the AutotuneAdaptor bean, as shown in Figure 2.1. The AutotuneController bean defines control strategies (such as learning the behavior of the target system or providing actions to amend abnormal situations).

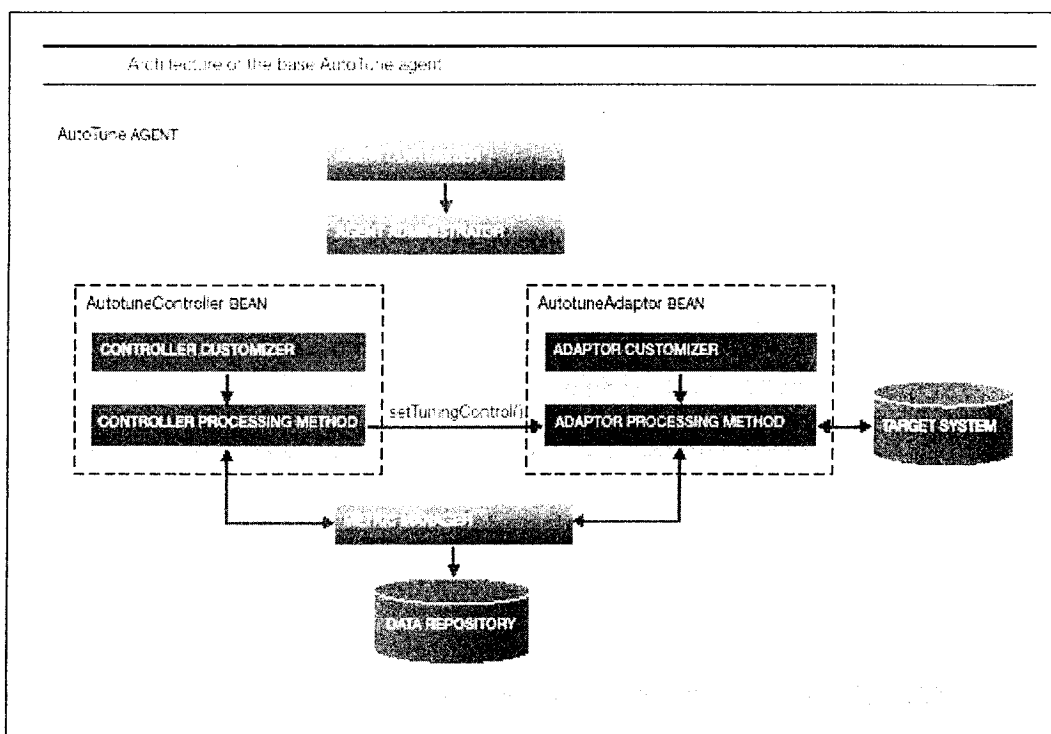


Figure 2.1: Architecture of the Bas AutoTune Agent [41]

Its Customizer GUI allows the system administrator to configure the control strategy in advance or on the fly. The AutotuneAdaptor bean interfaces with the target system to get the service level metrics and to set the tuning parameters. Another Customizer GUI is provided for the system administrator to manually set the tuning parameters (for example, for testing purposes, or when the AutotuneController is inactive). This decoupling allows the same AutotuneController to be used with a variety of systems, simply by choosing the appropriate AutotuneAdaptor to interface with the respective system. The execution of the AutotuneController and AutotuneAdaptor beans is managed by the AutoTune agent through the Agent Administrator. In particular, the Agent Administrator handles the timer facility and asynchronous event processing function, which allow the AutotuneController and AutotuneAdaptor to run autonomously, by periodically processing control functions and communicating with the target system. The AutoTune agent-level Customizer allows the system administrator to separately set the control interval (used by the AutotuneController bean) and sample interval (used by the AutotuneAdaptor bean), which can be different from each other. A set of AutotuneMetric classes is defined to represent the state/performance of

the target system (service-level metrics), the tuning parameters of the target system (tuning control metrics), and the parameters of the control strategy (configuration metrics). These metrics can be read or written by the AutotuneController or AutotuneAdaptor. They are managed as a collection by the AutoTune Metric Manager for interactions between the two component beans, and can also be selectively saved to a historical data repository. ABLE and the AutoTune architecture were designed to be extensible and to allow rapid deployment of agent based solutions. They indicate that their experience with using this infrastructure validates the usefulness of this architecture and the ABLE toolkit [27][41].

2.2.2 Autonomia.

Another related research project, which is concerned with the autonomic computing environment, is the Autonomia [36] research project. Autonomia provides dynamically programmable control and management services to support the development and deployment of smart (intelligent) applications. The AUTONOMIA environment provides the application developers with all the tools required to specify the appropriate control and management schemes to maintain any quality of service requirement or application attribute/functionality (e.g., performance, fault, security, etc.) and the core autonomic middleware services to maintain the autonomic requirements of a wide range of network applications and services. The researchers claim that they have successfully implemented a proof-of-concept prototype system that can support the self-configuring, self-deploying and self-healing of any networked application. The architecture consists of three main modules: Application Management Editor (AME), Autonomic Middleware Services (AMS), and Application Delegated Manager (ADM). The AME provides users with the software tools to describe the strategies to be used to achieve the required autonomic properties. The AMS provides a common set of autonomic services (e.g. self-configuring, self-healing, self-protecting, self-defining, etc.). The ADM is a software agent responsible to configure, deploy, run and maintain the autonomic properties of the application at runtime. The objective of this project is to automate deployment of mobile agents that have self manageable attributes. The architecture of Autonomia is based on two previous projects: Adaptive Distributed Virtual Computing Environment (ADVICE) and CATALINA – A Proactive Application Control and Management System [35]. The Autonomia environment provides application developers with all the tools required to specify the appropriate control and management schemes, deploy and configure the required software and hardware resources, run applications, and to provide on-line monitoring and management facilities to maintain desired autonomicity. The architecture of Autonomia is shown in the Figure 2.2 [36]. As a result of our contact with the researchers of this project we found out that it is no longer continued and we could not get a version of the implemented prototype. All the available information about this project is found through few published papers, which made it very difficult to deeply investigate the detailed architecture of this research work.

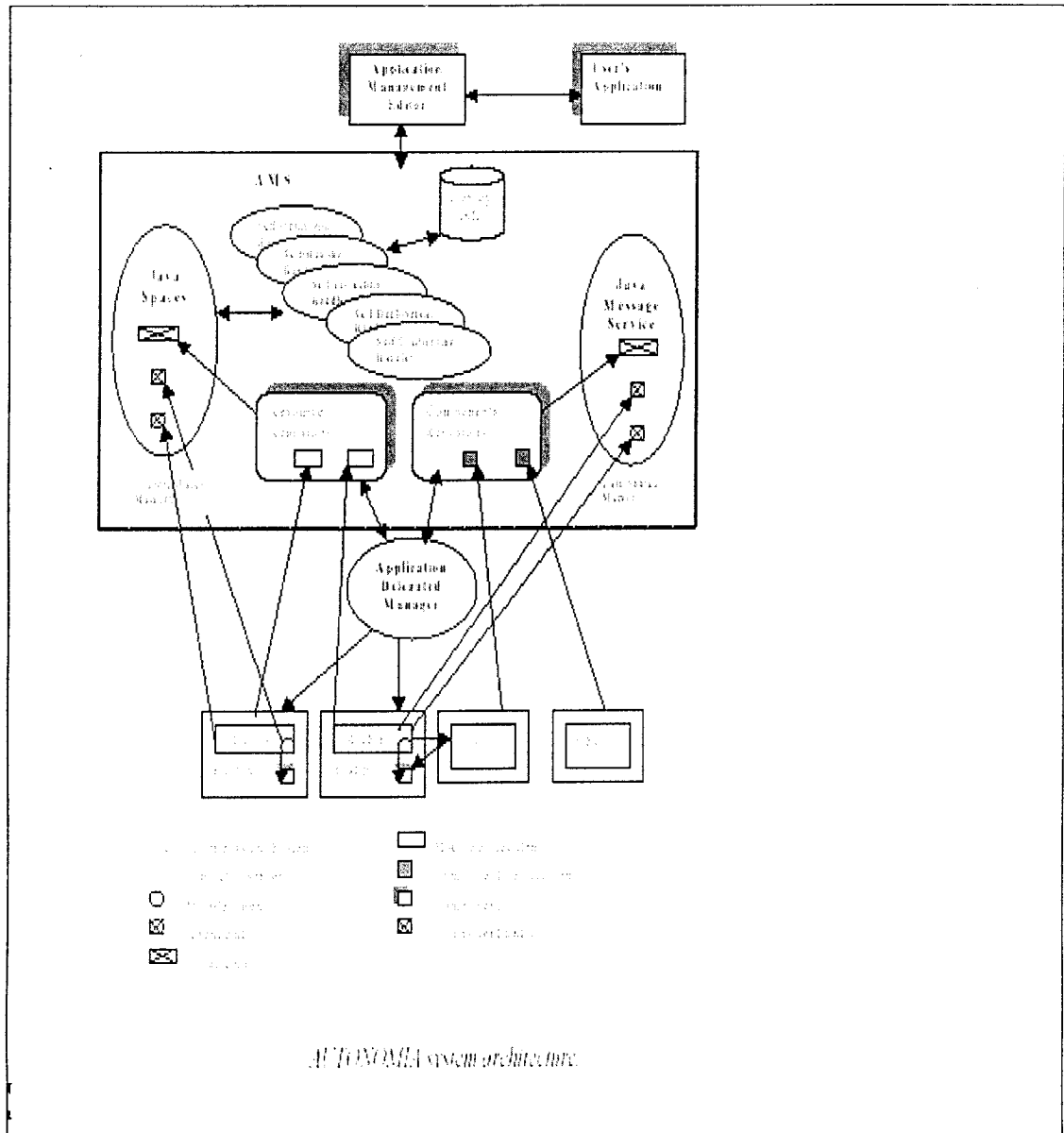


Figure 2.2: AUTONOMIA System Architecture [36]

2.2.3 An architecture for Autonomic Personal Computing Systems

In a research paper related to personal autonomic computing, the researchers [3] refer to some of the properties of autonomic computing as they are shared with personal computing such as ease of use and flexibility. The intention of their paper, is to identify the unique demands and opportunities of autonomic computing with personal devices. The ground rules that they seek to achieve, are the autonomic behavior of a personal computing system, personal computers (PCs) and their peers, networks, and servers not just the PC alone. They also presented some general considerations for an architecture that supports autonomic personal computing [11].

They introduced a categorization of autonomic function in terms of where it gets implemented. An Autonomic function can be implemented locally, drawing on locally maintained measurements and knowledge. It can be implemented among members of a peer group, sharing measurements and knowledge particular to that group. It can also be implemented using globally available network-resident resources, in which case, measurements and knowledge are maintained for all clients. In the most general case, autonomic functions are implemented in all three ways, with different functions having a preferred implementation.

The architecture of an autonomic system, including that of a personal computing system, begins with the general architecture for autonomic systems. The building block of autonomic systems is depicted in Figure 2.3, which shows the architecture of an autonomic element (AE). Each AE consists of an autonomic manager (AM) and a set of managed components. Each managed component is responsible for communicating its events and other measurements to the local AM. In turn, based on the input received from each managed component, the AM makes decisions taking into account its policy, facts, and rules (stored locally in a database) and communicates the directives and hints to the managed component.

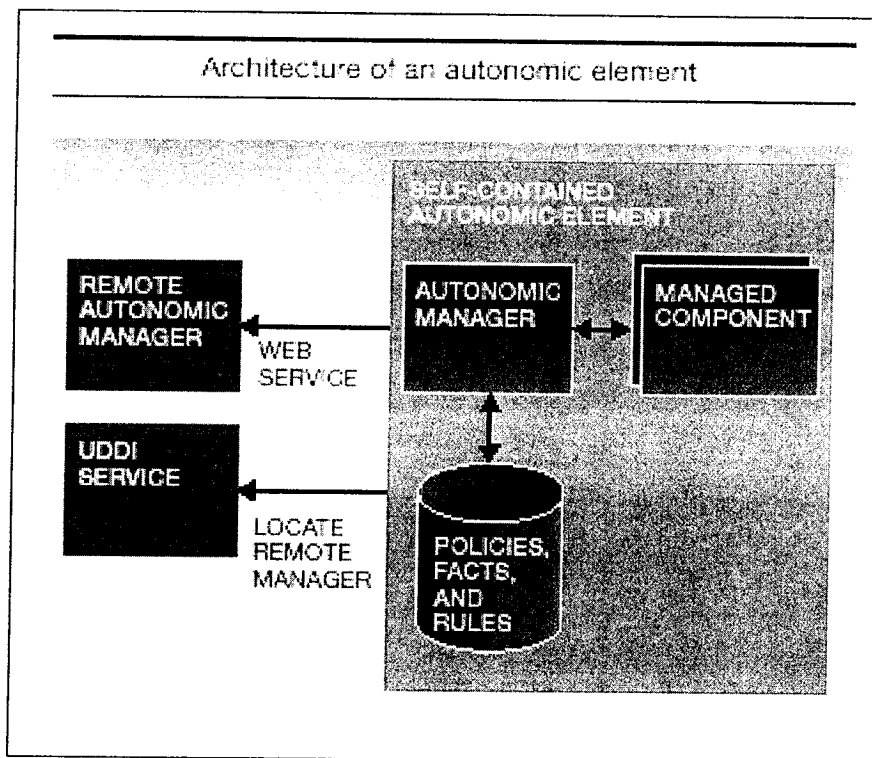


Figure 2.3: The Architecture Of An Autonomic Element [11]

The figure makes a distinction between self-contained autonomic behavior (within the large gray box on the right) and autonomic management involving explicit communications with a remote manager. The interfaces between an AM and its managed component are an important part of the architecture. This interface must be discoverable and dynamically bound so as to support self-configuration of autonomic systems; it must also be secure and private. The figure shows a remote autonomic manager implementing a Web service, located via the Universal Description, Discovery, and Integration (UDDI) service registry. We see Web services as a foundation technology because they provide standard ways to locate, communicate (via XML), compose, and interact with network-based services. But because personal systems are often mobile and occasionally disconnected, the interface must support a disconnected (offline) mode of use as well. Figure 2.4 shows the architecture of an autonomic system consisting of autonomic elements connected to one another at local, peer,

and network levels. Resources are shown as boxes, AMs as diamond shapes, peer groups as dashed ellipses, and physical resources, servers and clients as circles. Arrows represent the control exerted by AMs (e.g., S controls W). At the local level, there is a single AM (e.g., A) that is capable of independent decision-making. At the peer level, each AM interacts and shares knowledge and information with its peers and may act cooperatively, as though a virtual autonomic manager (e.g., W) is present. Only one AM is ever in direct control of a resource [11].

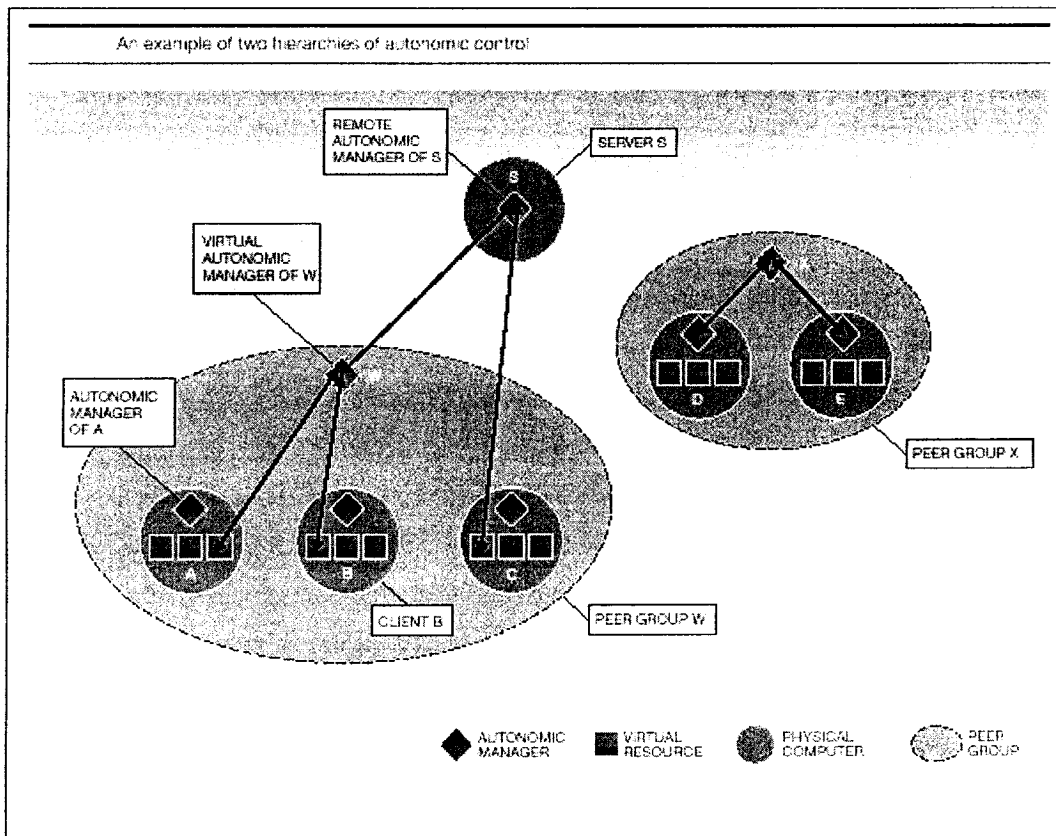


Figure 2.4: An Example of Two Hierarchies Of Autonomic Control [11]

2.2.3.1 Directions toward an autonomic framework

The researchers [11] define an autonomic framework that brings together disparate computing elements. The key elements of the autonomic framework are the autonomic manager and the elements to be managed. The goal of the framework is to specify the interfaces and protocols for elements to exchange information and data to enable collective

autonomic behavior. According to the researchers [11], to achieve this goal, we need to rethink the structure of the system and the application software (and the tools that help build them) so as to identify and expose relevant and accurate indications of the state of each element, and provide standard interfaces to affect an element state with minimal side effects. Each element will need an element-specific autonomic manager to monitor and control the element. This coupling of element and specific manager represents the lowest level of autonomic behavior. Elements may be isolated in virtual machines to limit undesirable interactions between them. Element specific autonomic managers will report to a system-wide autonomic manager in a standard way. The system-wide manager is responsible for achieving end-user goals in accordance with an established policy.

2.2.3.2 Autonomic managers

The elements under control of an autonomic manager must be observable and controllable. Current computing systems maintain a wealth of data about themselves in repositories and logs. Some of these data are redundant and confusing, and some are not even accurate. Thus the information relevant to decision-making is a challenge to obtain from these data sources. Similarly, many points of control exist, but their relationship to the desired behavior of the system is unclear [11].

2.2.4 An architectural blueprint for autonomic computing

In the recently published paper by IBM [1], IBM has presented an outline for the prospected architecture and framework of the autonomic computing during the coming phase. That architectural blueprint for autonomic computing is an overview of the basic concepts, constructs and behaviors for building an autonomic capability into an on demand computing environment. The blueprint also describes an initial set of core capabilities for enabling autonomic computing and discusses technologies that support these core capabilities. Each of these technologies is being developed or is undergoing further refinements. It also discusses industry standards, emerging industry standards and new areas for standardization that will make autonomic computing an open system architecture.

The blueprint states that the autonomic computing architecture starts from the premise that implementing self-managing attributes involves an intelligent control loop. This loop collects information from the system, makes decisions and then adjusts the system as necessary. An intelligent control loop can enable the system to do such things as:

- Self-configure, by installing software when it detects that software is missing
- Self-heal, by restarting a failed element
- Self-optimize, by adjusting the current workload when it observes an increase in capacity
- Self-protect, by taking resources offline if it detects an intrusion attempt.

Figure 2.5 illustrates that these control loops can be delivered in two different ways:

- The loop can be implemented by various combinations of management tools or products. In the diagram below, the three examples are the configuration manager, workload manager and risk manager. These tools use the instrumentation interfaces (for example, a Simple Network Management Protocol management information base [SNMP MIB]) provided by IT system components that make them manageable. This interface is referred to as the manageability interface in the diagram.

- A control loop can be provided by a resource provider, which embeds a loop in the runtime environment for a particular resource. In this case, the control loop is configured through the manageability interface provided for that resource (for example, a hard drive). In some cases, the control loop may be hard-wired or hard-coded so it is not visible through the manageability interfaces.

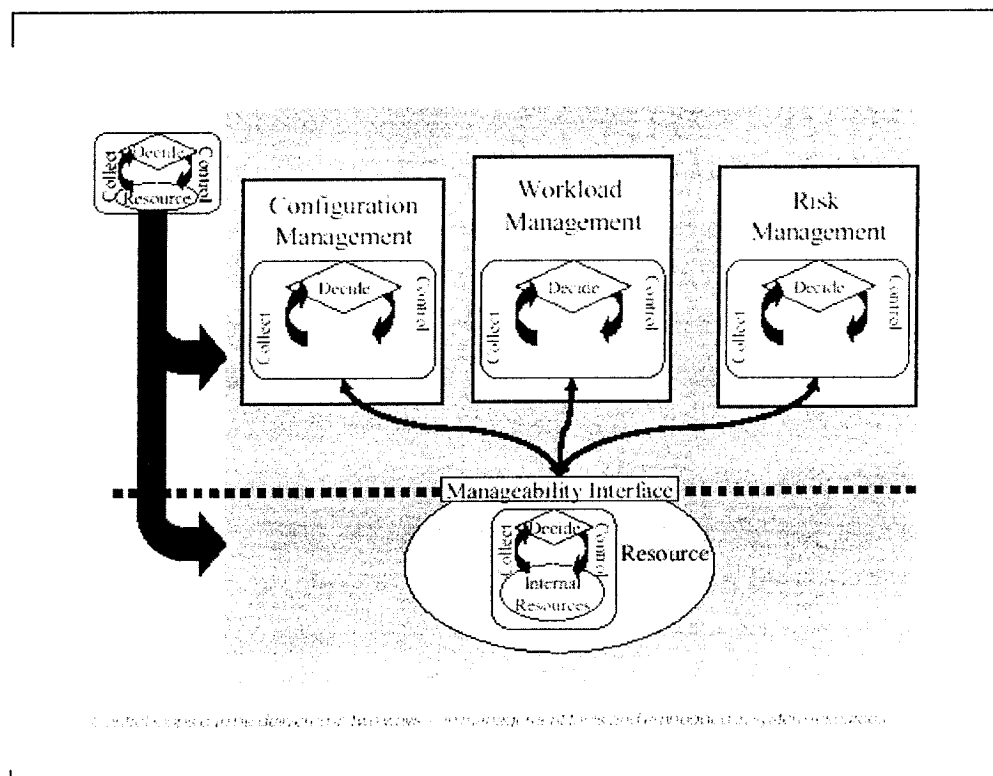


Figure 2.5: Intelligent Control Loops in An Autonomic System [1]

The architecture for autonomic computing defines three different layers of management. Each layer involves implementing control loops to enable the self-management in three different decision-making contexts or scopes:

1. The resource element context is the most basic, because its elements, networks, servers, storage devices, applications, middleware and personal computers manage themselves in an autonomic environment.
2. The resource elements are grouped into a composite resources decision-making context. These groups can be represented by a pool of servers that work together to

dynamically adjust workload and configuration to meet certain performance and availability thresholds; or they can be represented by a combination of heterogeneous devices, such as databases, Web servers and storage subsystems, working together to achieve common performance and availability targets.

3. At the highest layer, the composite resources are tied to the business decision-making context, such as a customer care system or an electronic auction system. The business solution layer requires autonomic solutions that will comprehend the optimal state of business processes based on policies, schedules and service levels.

These different management levels define a set of decision-making contexts that are used to classify the purpose and role of a control loop within the autonomic computing architecture. The architecture organizes the control loops into two major elements:

- 1- A managed element
- 2- An autonomic manager

A managed element is what the autonomic manager is controlling. An autonomic manager is a component that implements a particular control loop. The managed element is a controlled system component. There can be a single resource (a server, database server or router) or a collection of resources (a pool of servers, cluster or business application). The managed element is controlled through its sensors and effectors:

- The sensors provide mechanisms to collect information about the state and state transition of an element. To implement the sensors, you can either use a set of “get ”operations to retrieve information about the current state, or a set of management events (unsolicited, asynchronous messages or notifications) that flow when the state of the element changes in a significant way.
- The effectors are mechanisms that change the state (configuration) of an element. In other words, the effectors are a collection of “set ”commands or application programming interfaces (APIs) that change the configuration of the managed resource in some important way. The combination of sensors and effectors form the manageability interface that is

available to an autonomic manager. The architecture dissects the control loop into four parts that share knowledge:

1. The monitor part: provides the mechanisms that collect, aggregate, filter, manage and report details (metrics and topologies) collected from an element.
2. The analysis part: provides the mechanisms to correlate and model complex situations (time-series forecasting and queuing models, for example). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations.
3. The plan part: provides the mechanisms to structure the action needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.
4. The execution part: provides the mechanisms that control the execution of a plan with considerations for on-the-fly updates.

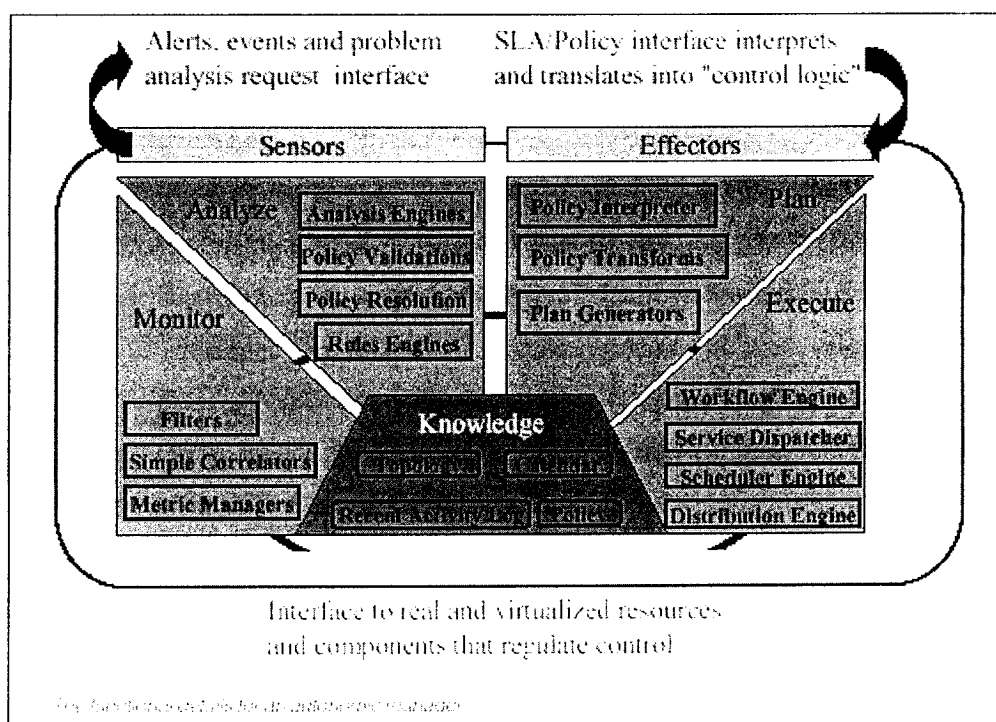


Figure 2.6: The Functional Details for An Autonomic Manager [1]

The four parts work together to provide the control loop functionality. Figure 2.6 shows a structural arrangement of the parts not a control flow. The bold line that connects the four

parts should be thought of as a common messaging bus rather than a strict control flow. In other words, there can be situations where the plan part may ask the monitor part to collect more or less information. There could also be situations where the monitor part may trigger the plan part to create a new plan. The four parts collaborate using asynchronous communication techniques, like a messaging bus [1].

Another important section that is relevant to this research area is a section, which describes an initial set of core capabilities that are needed to build autonomic managers. These core capabilities include:

1. **Solution knowledge:** From an autonomic systems perspective, lack of solution knowledge inhibits important elements of self-configuring, self-healing and self-optimizing. A common solution knowledge capability eliminates these complexities.

Common system administration: Autonomic systems require common console technology to create a consistent human-facing interface for the autonomic managers for the elements of the IT infrastructure.

2. **Problem determination:** The capability to be able to extract high quality data to determine whether or not a problem exists in the managed element.
3. **Autonomic monitoring:** Autonomic monitoring is a capability that provides an extensible runtime environment for an autonomic manager to gather and filter data obtained through sensors. Autonomic managers can utilize this capability as a mechanism for representing, filtering, aggregating and performing a range of analyses of sensor data.
4. **Complex analysis:** Autonomic managers need to have the capability to perform complex data analysis and reasoning on the information provided through sensors.
5. **Policy for autonomic managers:** An autonomic computing system requires a uniform method for defining the policies that govern the decision-making for autonomic managers.
6. **Transaction measurements:** Autonomic managers need a transaction measurements capability that spans system boundaries in order to understand how the resources of heterogeneous systems combine into a distributed transaction execution environment.

Technologies that deliver these capabilities will accelerate the delivery of autonomic managers that can collaborate in an autonomic system [1].

2.2.5 E-Model Framework.

In [10], Crawford and Dan describe a novel, flexible framework, e-Model, designed to address the runtime requirements of autonomic computing: on-line workload measurement, analysis, and prediction. The e-Model architecture was developed using a platform independent technology (XML and Java) to allow for maximum portability while also allowing for ease of integration with existing measurement and system management tools. The e-Model toolkit consists of a GUI based model builder tool, a data base deployment tool, a runtime tool, and an analysis tool. In addition to the toolkit, the e-Model design provides a runtime architecture which can be deployed directly without using any interaction with the GUI. The architecture is flexible enough to allow for incorporation with models of various complexity, including modeling techniques that require a hierarchical approach to attain reasonable accuracy. According to Crawford and Dan [10], the traditional focus on tooling has been on developing sophisticated offline or online tools capturing as much of the details of an environment as possible. Integrating these types of tools as runtime components of a system demands that we pay as much attention to the integration framework as to the models themselves. Ease-of-use continues to remain an important issue, however, not necessarily only as an issue in running the tool by a human operator but also as an issue in setting up the tool for system development/deployment. Their work in [10] introduces a flexible architecture for such a modeling framework, and demonstrates diverse usage scenarios for different objectives and/or environments and demonstrates how an e-model can play different roles in an autonomic environment. An autonomic system component typically monitors and reconfigures itself to comply with service level agreements* [17] on its usage, as established with the clients of this system. SLAs are established by clients with the service systems in order to receive a guarantee on various service level objectives, e.g., average response time for supported throughput level during certain time periods, availability of services, etc. The e-Model architecture is used during all phases of a SLA life-cycle: creation, deployment and runtime monitoring & enforcement. Different usage and roles that the an e-Model can play as defined in [10] are:

1. **SLA Advisor:** In order to establish an SLA, a client needs an understanding on its expected workload, perhaps predicated from past workload history. The data may

be available in predefined formats (i.e., as a file or database table). However, when such data are not available a-priori or an SLA needs to be renegotiated to reflect changing needs, the data may be collected from a running system. As the data collection occurs, workload models are built and new SLAs can be constructed by the SLA advisor.

2. **Risk Analyzer:** From a service provider perspective, during deployment and/or for making a commitment to a client SLA, the system needs to understand its available capacity, and analyze its risk in accepting this SLA. So, in this case an EModel tool framework can be used as a Risk Analyzer.
3. **SLA Monitor:** During actual service invocation, the e-Model framework can be used to measure and monitor runtime performance, and predict potential violations of service level objectives. This usage of e-Model framework is referred to as SLA Monitor. In addition sophisticated SLA monitoring may involve both computation of aggregated run-time parameters via metric composition (e.g., computing average from individual response times) and/or online prediction of future values of composed or component parameters.
4. **Resource Monitor:** The e-Model framework (by introducing the resources monitor) can also be used to further monitor an individual or a collection of resources, to watch for (current or predicted) problem states, e.g., high utilization, system bottlenecks, etc.
5. Finally, observed service performance data and/or observed customer workload can be used to adjust risk analysis, and/or to trigger renegotiation of existing SLAs.

The three main objectives in the e-Model system design are as follows:

1. **Ease-of-use:** The framework should be easy to use for both the model provider and the model user.

2. **Flexibility:** The e-Model runtime architecture should also be flexible enough to adapt to improving modeling techniques.
3. **Scalability:** The e-Model architecture should also be able to track and forecast multiple workloads of multiple types (of both service level and application request type) from possibly several remote locations.

In order to incorporate all of these design goals, the e-model provides a rich yet concise language to describe the workload to be estimated. For instance, that language is able to collect information that have answers to the following questions:

- 1- what quantities need to be measured and how often the online samples should be taken.
- 2- what are the important features in the time-series (workload) we are modeling (i.e. periodicity).
- 3- what is the input and output as well as necessary parameters for the models that are being used.
- 4- what type of prediction horizon and confidence interval should be used in data forecasting, and should we take any action based upon a prediction (i.e. alert or event generation).

The researchers [10] employed an architecture for an e-model toolkit which adheres to a more general architectural concept as shown in Figure 2.7. In this figure, they illustrate how user-input is transformed into XML [39][40] descriptors and database table entries and ultimately into Java runtime objects. This concept is used extensively in any container based object oriented architecture. The advantages of such a paradigm are clear:

1. The user is guided through a parameter input step (GUI based) and does not need to have any knowledge of XML or DB schematics.

2. At runtime the application developer only works with Java objects, again no knowledge of XML or DB is required.
3. The data is preserved if the system experiences some failure while the eModel toolkit is tracking data, the data has been preserved in a DB so that a model which improves with historical data need not start from default initial conditions which may result in deteriorated accuracy.

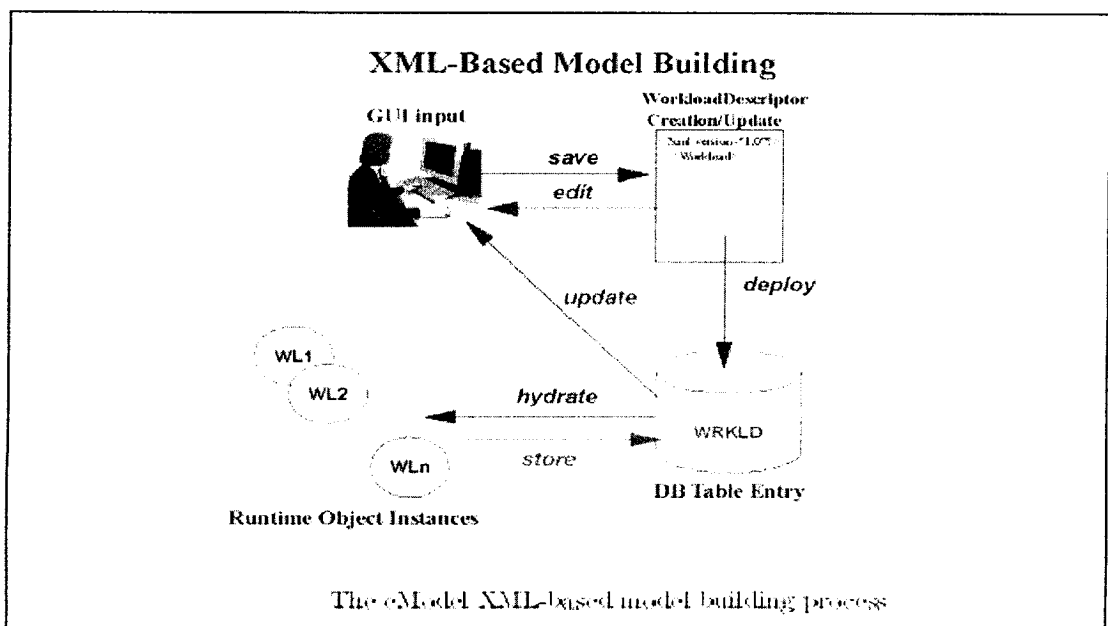


Figure 2.7: The eModel XML-Base Model Building Process [10]

Finally the authors (C.H Crawford & A. Dan [10]) have presented several examples that illustrate the e-Model as a capacity planning tool as well as an augmentation to autonomic system management in an effort to highlight the technological gaps that the eModel framework is capable of bridging. Existing complementary model building tools can be integrated into the e-Model architecture such as the Agent Building Learning Environment (ABLE) Toolkit and that will be expected to ease the process of model building [26].

2.2.6 AutoMate

The overall objective of the AutoMate [31] project is to investigate key technologies to enable the development of autonomic Grid applications that are context aware and are capable of self-configuring, self-composing, self-optimizing and self-adapting. Specifically, it will investigate the definition of autonomic components, the development of autonomic applications as dynamic compositions of autonomic components, and the design of key enhancements to existing Grid middleware and runtime services to support these applications. Specific issues addressed include:

1. **Definition of Autonomic Components:** The definition of programming abstractions and supporting infrastructure that will enable the definition of autonomic components. In addition to the interfaces exported by traditional components, autonomic components provide enhanced profiles or contracts that encapsulate their functional, operational, and control aspects. These aspects enhance the interfaces to export information and policies about their behavior, resource requirements, performance, interactivity and adaptability to system and application dynamics. Furthermore, they encapsulate sensors, actuators, access policies and a policy-engine. Together, aspects, policies, and policy engine allow autonomic components to consistently configure, manage, adapt and optimize their execution.
2. **Dynamic Composition of Autonomic Applications:** The development of mechanisms and supporting infrastructure to enable autonomic applications to be dynamically and opportunistically composed from autonomic components. The composition will be based on policies and constraints that are defined, deployed and executed at run time, and will be aware of available Grid resources (systems, services, storage, data) and components, and their current states, requirements, and capabilities.
3. **Autonomic Middleware Services:** The design, development, and deployment of key services on top of the Grid middleware infrastructure to support autonomic applications. One of the key requirements for autonomic behavior and dynamic

compositions is the ability of the components, applications and resources (systems, services, storage, data) to interact as peers. Furthermore the components should be able to sense their environment. In this project, the authors extend the Grid middleware with:

- (1) a peer-to-peer substrate.
- (2) context aware services.
- (3) peer-to-peer deductive engines for composition, configuration and management of autonomic applications.

An active peer-to-peer control network combines sensors, actuators and rules to configure and tune components and their execution environment at runtime and to satisfy requirements and performance and quality of service constraints.

The overall research objective of the AutoMate [31] project is to develop and deploy the AutoMate framework for enabling autonomic Grid applications. The used technical approach is built on three fundamental concepts:

1. Separation of policy from mechanism distilling out the aspects of components and enabling them to orchestrate a repertoire of mechanisms for responding to the heterogeneity and dynamics, both of the applications and the Grid infrastructure. The policies that drive these mechanisms are specified separately. Examples of mechanisms are alternative numerical algorithms, domain decompositions, and communication protocols; an example of a policy is to select a latency-tolerant algorithm when network load is above certain thresholds.
2. Context, constraint and aspect based composition techniques applied to applications and middleware as an alternative to the current processes for translating the application's dynamic requirements for functionality, performance, quality of service, into sets of components and Grid resource requirements.

- Dynamic, proactive, and reactive component management to optimize resource utilization and application performance in situations where computational characteristics and/or resource characteristics may change. For example, if adaptive mesh refinement increases computational costs, we may negotiate to obtain additional resources or to reduce resolution, depending on resource availability and user preferences.

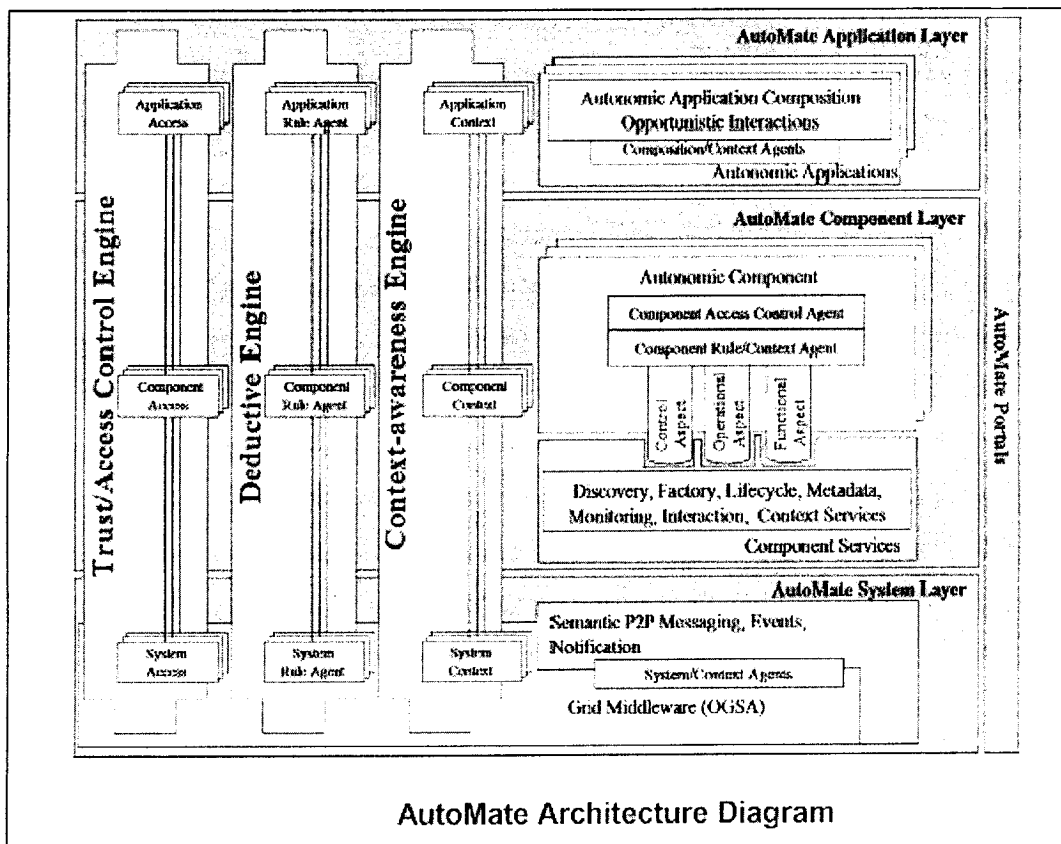


Figure 2.8: AutoMate Architecture Diagram [31]

Building on these fundamental concepts, AutoMate addresses fundamental issues and provides key solutions in the autonomic formulation, composition, and runtime management of applications on the Grid. A schematic of the overall architecture is presented in Figure 2.8. AutoMate builds on the emerging Grid infrastructure and extends the Open Grid Service Architecture (OGSA)[34]. AutoMate is composed of the following components:

1. **AutoMate System Layer:** The AutoMate system layer builds on the Grid middleware and OGSA and extends core Grid services (security, information and resource management, data management) to support autonomic behavior. Furthermore, this layer provides specialized services such as peer-to-peer semantic messaging, events and notification.
2. **AutoMate Component Layer:** The AutoMate component layer addresses the definition, execution and runtime management of autonomic components. It consists of AutoMate components that are capable of self configuration, adaptation and optimization, and supporting services such as discovery, factory, lifecycle, context, etc. (which builds on core OGSA services).
3. **AutoMate Application Layer:** The AutoMate application layer builds on the component and system layers to support the autonomic composition and dynamic (opportunistic) interactions between components.
4. **AutoMate Engines:** The AutoMate engines are decentralized (peer-to-peer) networks of agents in the system. The context-awareness engine is composed of context agents and services and provides context information at different levels to trigger autonomic behaviors. The deductive engine is composed of rule agents which are part of the applications, components, services and resources, and provides the collective decision making capability to enable autonomic behavior. Finally, the trust and access control engine is composed of access control agents and provides dynamic context-aware control to all inter-actions in the system.

In addition to these layers, AutoMate portals provide users with secure, pervasive (and collaborative) access to the different entities. Using these portals users can access resource, monitor, interact with, and steer components, compose and deploy applications, configure and deploy rules, etc [31].

2.2.7 Business Workload Manager Prototype (BWLM).

The Business Workload Manager (BWLM) Prototype [9] is a technology presented by IBM, which enables the instrumentation of applications with Application Response Measurement (ARM) in order to monitor the performance of transactions across a distributed environment. This ARM-based performance information is used by BWLM to monitor and adjust the allocation of computing resources on an ongoing, split-second basis. Planned functions include the ability of BWLM to detect changes in its environment and decide which resources (system, network, load-balancing patterns) to adjust in order to enable a network of systems to meet end-to-end performance goals. When middleware (or, in this prototype, an application) is instrumented with ARM, it is expected to take advantage of products such as BWLM and participate in IBM's autonomic computing initiative. The prototype allows one to observe and build upon the instrumented application using the ARM 4.0 (pre-approval version) standard to handle workload management for better transaction flow across systems and applications. This technology will provide significant value in understanding response times and transaction flow for the following [9]:

- Improved service-level management based on performance policies
- Determining where transactions hang.
- Active workload management for better capacity use.
- Understanding bottlenecks for better capacity planning.

The prototype demonstrates instrumentation of an application (in this case, a PlantsByWebsphere EJB) using ARM APIs, which could otherwise have been achieved by instrumenting a middleware such as WebSphere. The use of service classes to define performance policies is also shown. The pre-approval ARM 4.0 standard supports both C and Java applications and will allow developers to instrument applications so that they collect performance data. This technology is intended to drive the first significant ARM instrumentation in commercial middleware [9]. The basic prototype offers the following features:

- Administrative application.
- Management server and BWLM agent for collecting ARM data.
- Simple reporting.
- Server class reporting.

- Service class drill-down reporting.
- High-level server statistics.

2.2.7.1 eWorkload Management overview

eWLM, which is part of the BWLM toolkit, allows you to gather and monitor performance statistics across a network of systems in a cross-platform environment. Figure 2.9 shows a sample environment to demonstrate how eWLM monitors transactions. In this sample environment, a light grey box indicates a system; in this case, a pSeries. All of the systems in this environment use the AIX platform on a pSeries server and have been set up to be managed by eWLM. The inner box (purple in color or dark grey in non colored version) indicates an application environment that runs on the server. All transactions begin in the Apache application environment. Once a transaction is submitted, it is classified into a service class and, optionally, a report class. In this example, the transaction is classified into a service class and, optionally, a report class when it enters the Apache application environment, since this is the first application environment that can recognize that a transaction has been submitted. The transaction may continue to the WebSphere application environment or the Local ARMed application environment in order for it to be completed. eWLM does not determine how the transaction will be processed; however, it does monitor the work as it flows through the environment.

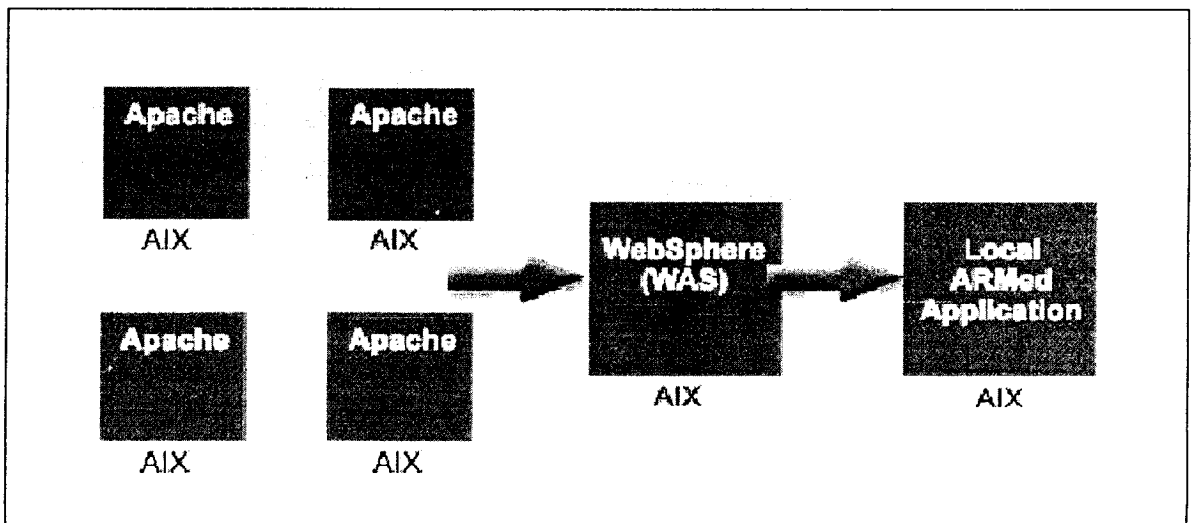


Figure 2.9: eWLM Sample Environment [9]

2.2.7.2 Management environment

The eWLM is designed to monitor response time and other goals that are set across multiple systems networked to a management server. A server and its networked systems form a domain. Using the eWLM Administration Console on a PC, one can define performance objectives, called policies, for various classes of users and applications. The preferred policy can then be activated across the entire domain. A domain can include different machine types and operating system platforms. One can create multiple domains and manage each domain's workload by using a single policy that is appropriate for that domain. The management server identifies which systems belong to each domain. An eWLM domain is a group of one or more systems networked through a management server. The management server compares the goals in the active policy with real-time analysis of the data streaming from the various servers to achieve an end-to-end view for reporting and analysis. The user interface lets one monitor and report the performance information [20].

All the work that runs in the domain is divided into workloads. For example, all the work that is created by a development group, or all the work that is started by an application or that resides in a subsystem could be a workload. Within a workload, one groups the work that has similar performance characteristics into service classes. One creates a service class for a group of work with similar performance goals. A policy can have one or more workloads, and a workload consists of one or more service classes. The service class specifies how to handle incoming work using a goal. There are four types of goals that one may specify. They are: Percentile response time, average response time, velocity, and discretionary. Usually a service class is created for a group of work with similar performance goals. For example one can create a service class for long-running work and a service class for short-running work. You will specify the service class to use for each type of work when you create an application environment's classification rules. An application environment can represent a programming environment such as middleware (for example WebSphere), an important application (for example Apache Web Server). The application environment uses filters for assigning the product's transactions to service classes and report classes. Each product

defines its own filters by using the ARM API. The eWLM Administration Console also provides filters that can be used with any application, such as System Name [20].

We have installed the trial version of eWLM prototype provided by IBM [9] to do some testing for the previously described features. Some of the toolset architectural model were really inspiring for our proposed architectural model. However, the eWLM is only a good tool for system monitoring and performance reporting, since the ARM interface merely allows data collection and classification. In fact the ARM interface itself imposes a limitation in the toolset usage, since each of the monitored systems should make an explicit call to the ARM APIs inside the application code in order to log the application transactions by the managed server. IBM claims that the full version will be completely different from the provided prototype and will contain more practical features and efficient management GUI but unfortunately the full version has not been released until the writing of this work. With reference to the current prototype we can conclude that it does not yet fully satisfy the prospective of an autonomic system.

2.2.8 Summary and Discussion of Related Work

In this chapter we summarized the research works and products that were found in the literature on autonomic computing technology. Some of the research works like *Autonomia* [36], *AutoMate* [31], and the *E-Model* framework [10] presented useful autonomic software design models. Other research works such as the *AutoTune* agents [41] took a different approach to present a product specific autonomic solution (i.e. the auto-tune agents for the apache server). On the other hand some vendors like IBM provided useful research work directions in the autonomic computing field by presenting useful publications such as the architectural blueprint for autonomic computing [1] or produced tools like the *BWLM* [9] to help in the management of an autonomic system.

The *AutoTune* research work proposed an efficient model for managing the Apache WebServer dynamically. However, the CPU and Memory utilization range needs to be specified statically and supplied by the administrator. Actually, because the Auto-tune agent does not have a more general overview of the overall system/machine resources in which it resides and is not aware of the other applications/servers that might share the same resources, it will not be able to determine the best range of resource utilization in which it can operate

dynamically. This information has to be supplied by the system administrator in order to be able to distribute the available resources consistently among all the running systems. Again, every time the administrator decides to redistribute the amount of resources consumed by the server in order to give more resources to a higher priority system, he will have to do this job manually. Certainly, such a job is a tedious one that requires continuous and close monitoring for all the systems in order to make the best utilization of all the utilized resources. Another important issue regarding this model is the inability to express goals or policies at different managerial levels. For example, the administrator should be able to specify the policies at one level from the business point of view and again at a lower level from the technical point of view which should also server the business level policies.

With respect to the Autonomia research paper, it does not provide enough details about the project implementation or detailed design. However we can conclude that the scope of the research does not cover the autonomization of the already running or used systems, programs, and packages in both the IT and Non-IT industries. This model is only exposed to the newly created applications that should implement or use the Autonomia model in order to design and implement an autonomic system. Such an assumption imposes a limitation on the usage of this model. In addition, this model also has a management scope problem since it is not fully aware of the entire system environment or the other running systems and application in order to make accurate corrective decisions when needed.

With regards to the personal autonomic computing research work which presented a hypothetical model for both autonomic personal computing and autonomic systems in general, the authors emphasized the importance and the need for the autonomic manager that will be able to manage all the allocated resources for a specific system. The aim of the autonomic manager in their work is to provide a sort of centralized management for a group of managed elements (such as Personal Computers). As mentioned in their manuscript the management could be at different levels (e.g. Local Manager, Group Manager, and Remote Manager). The researchers do not specify whether these managers take care of more than one management activity at the same time or not (i.e. handling security updates, configuration updates, etc at the same time) and if they do, it is not clear whether they coordinate these activities at the same time. The researchers also did not propose a detailed

approach for the implementation of the autonomic managers within the personal computing environment nor provide enough details to the proposed model.

The blueprint for autonomic computing research work presented very useful directions and guidelines for any researcher who wishes to propose a model for the autonomic computing since it provides a definition for most of the basic features that should be included in any autonomic system. They also provide the control loop model, which is one of the core components of any autonomic system. Actually, we have partially integrated this model in our proposed architecture of the autonomic property manager as explained in the design and architecture chapter of our research work.

The e-model toolkit presented in the research work, can play different roles, which range from SLA advisor to system resource monitor. However this tool is a passive monitoring tool. In other words it cannot take any corrective actions based on events or condition evaluation. So it can only do system monitoring and reporting which of course imposes a limitation in the tool usage.

In a very similar manner the BWLM prototype provides a comparable functionality to the one provided by the e-Model toolkit. So again this tool can only do system monitoring and reporting and is not able to take any corrective actions.

The AutoMate research project provides an infrastructure or a sort of middleware for a set of autonomic services on top of the Grid services infra structure to enable the development of autonomic Grid applications which are context aware and are capable of selfconfiguring, self-composing, self-optimizing and self-adapting. In other words it attempts to facilitate the development of autonomic applications as dynamic compositions of autonomic components, and the design of key enhancements to existing Grid middleware and runtime services to support these applications. Even though this research work tackles the autonomic computing field from an approach that is different from our approach, it provides some useful ideas concerning the layering of the different services provided by the autonomic middleware.

In Summary, each research work tackled the autonomic computing dilemma from a different approach and each of them has really presented new and useful ideas to the field of autonomic computing and to this work. However, none of them provide a complete solution to the problem yet. This fact implies that there are still more challenges on the road towards a

flexible model for the autonomic computing and this is what this research aims to contribute in while following a different perspective from those of the above mentioned research efforts

Chapter 3. Proposed Architecture for the Autonomic Property Manager

3.1 Introduction

The purpose of this chapter is to present a detailed description of the proposed system architecture for a generic autonomic property manager. This architecture was validated through the Resource Optimization Property Manager prototype implementation, which was used in all the experimental work for this research (described in detail in chapter 4). Since the architecture is a means of achieving the requirements of the system [8], the motive behind the presented architecture is to satisfy the autonomic system requirements. This architecture is consistent with our new perspective for the notion of autonomic computing. In our intended design we were seeking an approach that would simplify some of the complexities imposed by the nature of the autonomic computing systems. The notion of the property manager was seen to guarantee the knowledge and professional expertise of specialization. What this means is that each property manager will be responsible and capable of maintaining one of the autonomic properties appropriately by embodying the required knowledge and tools to handle it. In this way also a group of property managers, each maintaining its respective role, can cooperate together to form an autonomic system, which is capable of maintaining the whole set of autonomic properties. The overall setup would finally lead to a self management-capable system. One important issue that this proposed architecture took care of, is the ability of integrating the already developed and existing software products and solutions into the policy definition of any of the autonomic property managers which in turn would avoid going into the process of reinventing the wheel. Hence, we were also concerned with providing the most generic autonomic property manager architecture in which any of the property managers can fit by replacing some of the modules that add the functionality and specialization of each property manager. In the next section we will give a description for our overall conceptualization of the whole autonomic model with its different levels. We will also specify the level, which our architecture model targets and that will provide a good reasoning for the methodologies behind this architecture

3.2 Autonomic Levels

First we present a diagram for the Autonomic Computing model, on which we built most of our assumptions for this research work. In Figure 3.1 we assume that there are three different hierarchal levels at which the autonomic computing properties could be

implemented. The very first (lowest) level, which is partially out of the scope of this research, is mainly concerned with the implementation of Autonomic Systems/Managers that could autonomically manage a certain resource or system. The AutoTune agent [41], which was implemented to manage the performance of the Apache web server, is one example for the scope of this level. In this example the AutoTune manager was only concerned with handling the optimization property of the Web Server; however we could have another manager at the same level, which is concerned with more than one autonomic property. In general, at this level the management is concerned with low-level details of the managed system. At this level also the autonomic manager is aware of all the system handling parameters, sensors, and effector functions through which it can manage the system efficiently. At the second level, which is the focal point of this research, the Autonomic property manager is represented. At this level each property manager is concerned with managing all the registered subsystems with respect to the property it represents. For example, an autonomic property manager, which represents the optimization property, (i.e. The Optimization Autonomic Property Manager) is only concerned with managing all the supervised subsystems in the best optimum way with respect to a predefined set of policies. It is mainly concerned with the supervision, and enforcement of all the optimization related policies. So we could have as many properties managers as we need, having each property manager representing one single autonomic property. For example, we could have a Security Manager, a Recovery Manager, and an Optimization Manager. This model gives the flexibility of representing any newly added property according to the need of the autonomic model, which can be achieved by creating a new property manager to represent the new property. The third hypothetical level, which is also out of the scope of this research, is the level at which the global manager resides (that is the brain of the autonomic environment). Mainly this manager is concerned with managing and solving all the problems that could not be handled by the other autonomic managers. Other managers also reference it as a consultant to some problems, which could not be solved by any of the property managers. The exact roles of the global manager is out of the scope of this research as mentioned earlier as it needs further research by itself; however assuming its existence is necessary in order to coordinate between the different property managers and to provide a conflict resolution strategy when needed. Actually the property managers should be capable of escalating

problems directly to system administrators or invoking certain interfaces that are provided by the Global manager to trigger certain actions or to provide the type of desired help. Hence, by reaching this point, we have defined the scope and level of operation (Level 2) that the autonomic property manger will cover in constructing a fully autonomic and dynamic system model.

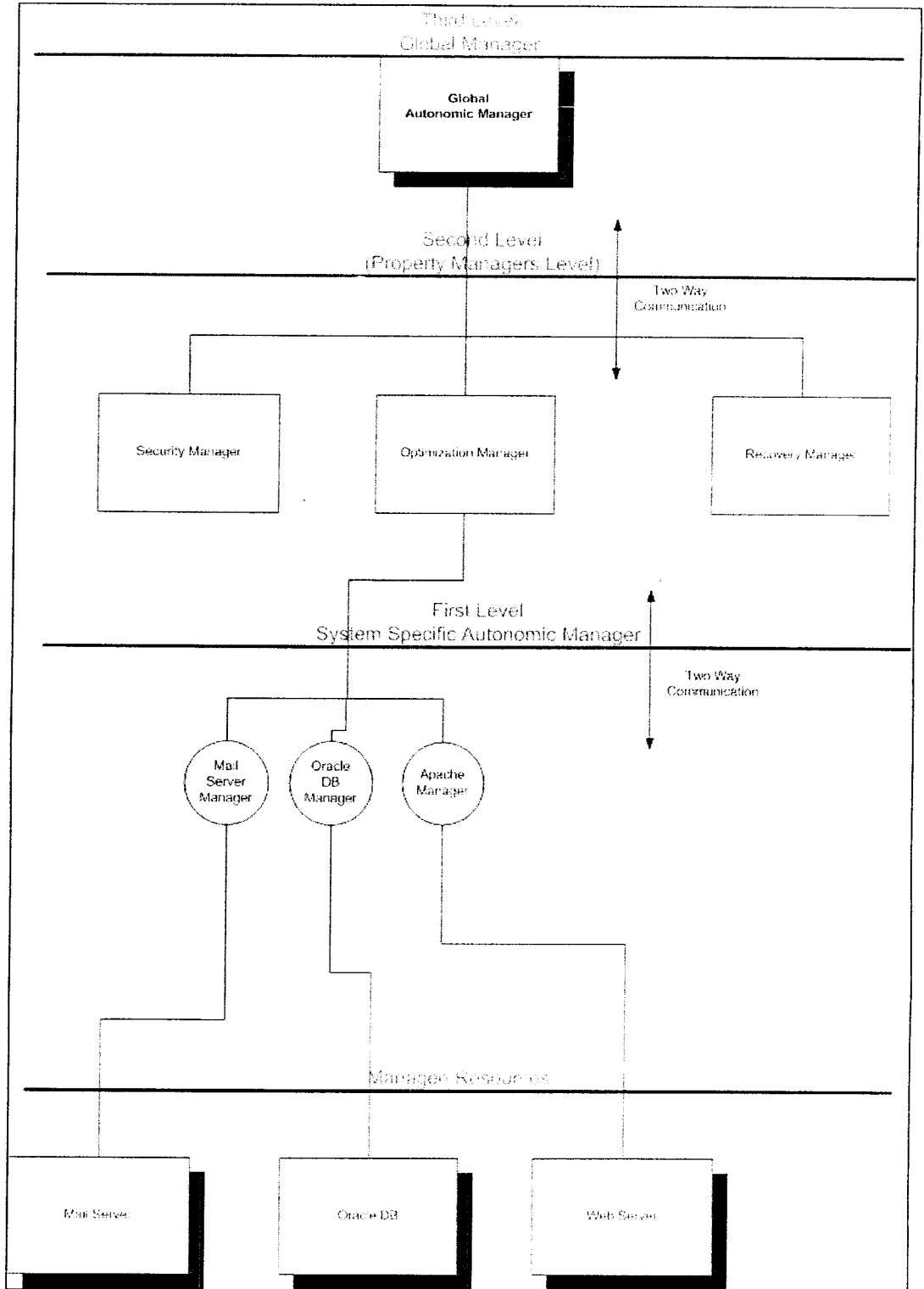


Figure 3.1: Levels Of Autonomic Representation

3.3 Architecture Essence.

3.3.1 Top Down Architecture Description.

Throughout our architecture description, we will use a top-down design approach. Hence, we first start by describing the system from the user's point of view. Starting from the GUI description level including all the related use cases, then we gradually move to the subsequent layers on which the property manager core functionalities are built. Our architecture is simply based on the client-server model (i.e. we have a client side and a server side). The server side is the place where all the property managers are placed and running, and that could be a dedicated machine or a machine hosting and running other applications. The client side is where the controlled application/system or the autonomic manager resides i.e. where the managed element exists. So we will start by describing the functionalities that the system administrator needs to perform in order to easily manage and control an autonomic system and this is illustrated in the next section.

3.3.2 GUI Description

The use case diagram shown in Figure 3.2 describes the major functionalities that are provided to the autonomic system administrator. Some of these functionalities include adding a new system to be managed by one of the property managers. Defining a new system entails that the system administrator will have to provide a set of certain parameters that are related to the new system and are of great importance to the property manager. An example for such kind of information is the system name, policy file name and location, system priority, system address, run frequency, and optionally a system description. The snapshot shown in Figure 3.3, presents the GUI used by the system administrator to define a new system, which is to be monitored by one of the property managers. The system name defines the unique name for the system, whereas the system address indicates the place at which the system resides and that could simply be an IP address or a machine name, which is defined within the network. So all we care about in the provided system address is that it should be reflecting a machine name or an IP address that really exists in the corporate network in which the system will run.

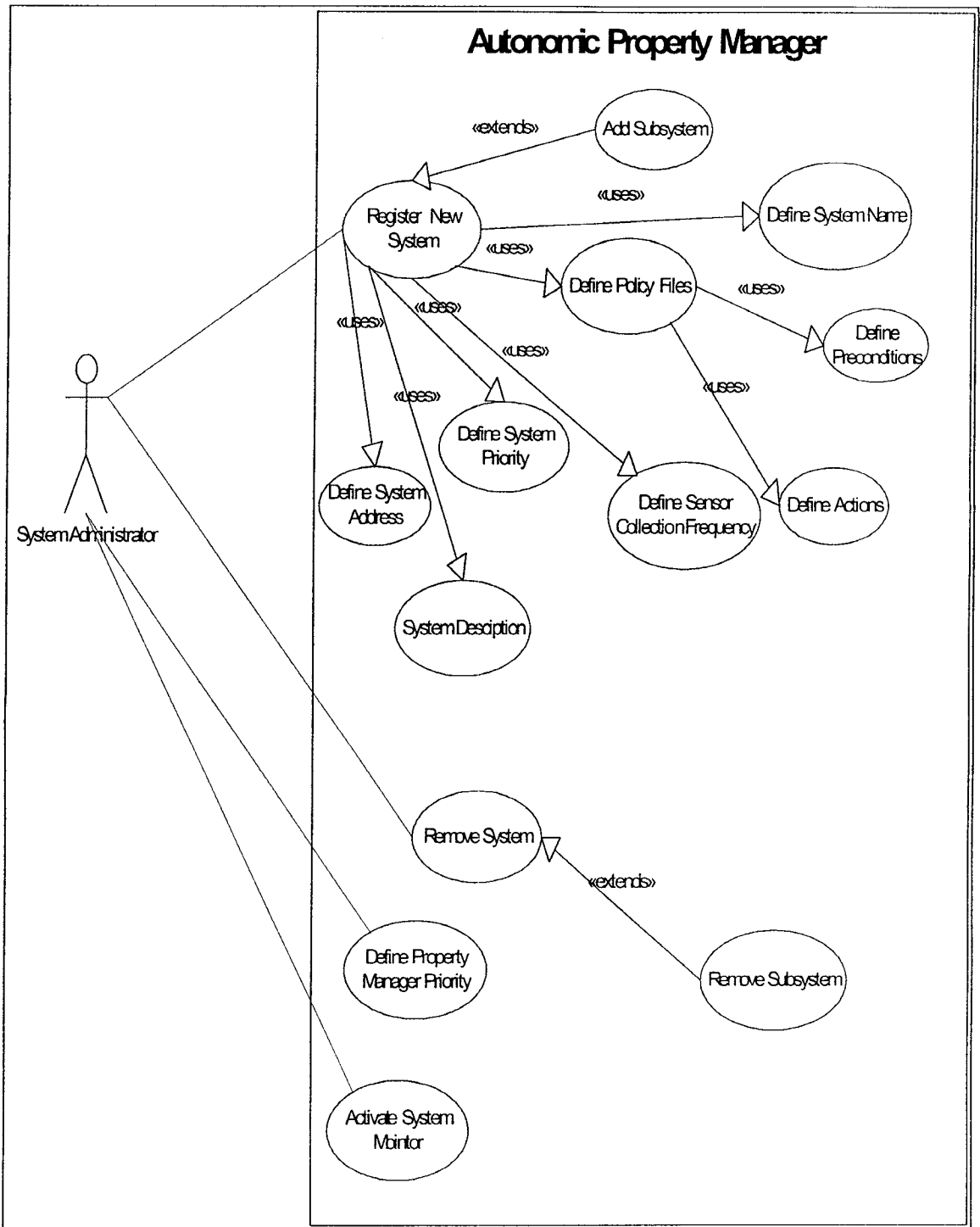


Figure 3.2: Autonomic Property Manager Use Case

The policy file fields are provided to supply the physical path of the system policy files and their names. A detailed description of the policy files purpose and structure will be given in the subsequent sections. A priority level has to be assigned to the system in order to specify the order in which the system will receive any shared event notifications among the different systems. A detailed description for the event notification mechanism will also be given in the up coming sections. The run frequency specifies the number of times that the sensors, inside the main system policy definition files, will run per time unit. For example the system shown in Figure 3.3, will run 10 times each minute. The unit could be specified as hours, minutes, days, months, etc. Finally a system description section is provided to write down any important notes or description about the defined system. Figure 3.4 provides another snapshot for the GUI window, which is used by the system administrator during a new subsystem definition. This window is very similar to the one used for the main system definition, however some of the fields such as the run frequency and system priority are removed since subsystems could only be triggered by main systems and they have the same priority assigned to their main systems. Additionally the parent system field is added in order to define the parent system or the owner system of the newly defined subsystem. The system administrator also provides a definition for the peak times of each system and these are the times in which the running system are having high loads, consequently they are critical operation times in which the system cannot tolerate to give some of its used resources. In fact during its peak time operation a system might need extra resources. This type of information is particularly important for the resource allocation manager to check out during the resource allocation look up process, which will shortly be explained in details. Figure 3.5 provides a snapshot of the GUI provided for the system administrator through which he/she can define the peak operation times for each system. All the provided data for each system are stored permanently in a database server so they can be retrieved or modified at any time.

Figure 3.3: Add New System GUI Snapshot

Figure 3.4: Add New Subsystem GUI Snapshot

System Name	From Date	To Date	From Time	Time Time
AUC Web Site	01-Apr-2004	01-Apr-2004	09:00 Am	11:00 Am

Figure 3.5: System Peak Times Definition GUI Snapshot

The ER diagram and the tables used in the system database are shown Figure 3.6. The figure shows the field names, defined tables, and their relationship. In our prototype implementation we used the MS database SQL server to implement the mentioned tables and store all the necessary data. Using a database server is more practical than using XML files to store the given data as data manipulation, processing, and retrieval is much easier using a database server. The database server also provides a higher level of security, and remote access mobility and above all it is more reliable with maintaining the data. Also as the number of registered systems increase, it will be much easier and faster to handle a large amount of data by a database server rather than by storing them in a large number of files. Finally Figure 3.7 presents a snapshot for the administration console where a full tree definition for a registered website to be managed by the optimization property manager is shown in the left pane. We can easily tell the system hierarchy depicted in the left pane by the simple interface provided to the system administrator to manage all of the systems that are managed by an autonomic property manager. The figure shows that the website system consists of three subsystems, which are an http server, application server, and a database server and they are displayed in a hierarchical tree shape.

Resource Optimization Property Manager For Autonomic Computing

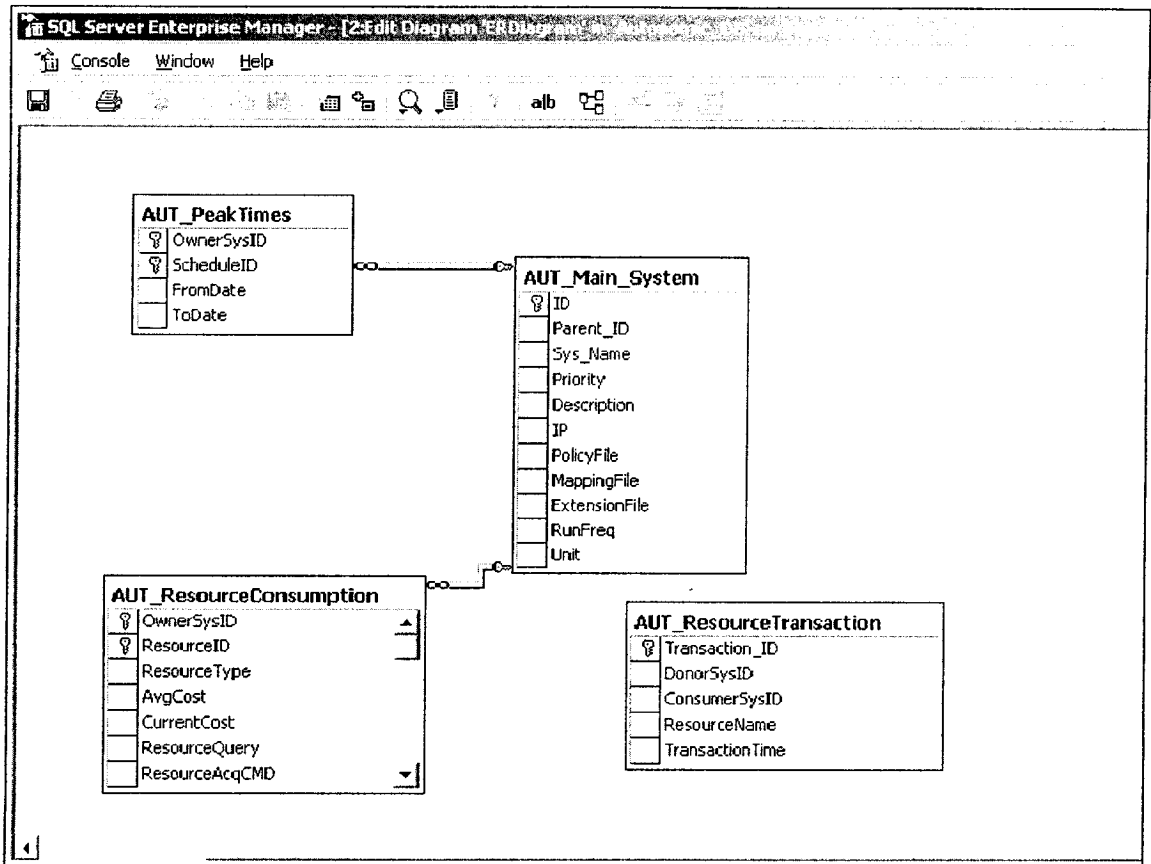


Figure 3.6: DB ER Diagram

Admin Console				
File Tools Help				
	Name	Address	Status	Description
Autonomic Property Managers	AUC Web Site	192.168.0.1	Running	Represents the Univers
Resource Optimization Manage	Web Site			
	HTTP Server			
	Application Server			
	DB Server			
Security Manager				

Figure 3.7: Admin Console Snapshot

3.3.3 Sensors (Actuators) and Effectors

The AMTS policy engine, which is a part of the ETTK [15], uses two major notions throughout the system policy definitions, which are known as sensors and effectors. We will provide a brief definition for these two notions as prescribed in the AMTS manuals [13-15], since we will be using the same terms throughout our architecture description. Sensors provide an autonomic manager with the ability to determine what is occurring around it. They connect the manager to the underlying element it is managing, as well as to other autonomic elements and other resources in the environment. Effectors allow the autonomic manager to control things by coupling the managed element to its controls, as well as permitting the element to interact with other autonomic elements and other resources in the environment. There is only one type of effector that may return a fact.

Sensors (Actuators) do not intentionally change the state of the element they are sensing. Sensing may create a load or other measurable effect on the managed element but should not cause a change in other ways. Sensors fall into two broad categories, active and passive. Passive sensors return a fact when they are called. Passive sensors effectively act as proxies to the actual data source, doing any work needed to access and format data and presenting it back to their callers as autonomic facts. Active sensors (alternatively called polling sensors), as their name implies, operate independently. They schedule timers so they can poll underlying resources, or are subscribed to an external source of events. Active sensors evaluate the current value they are sensing, and depending on the details of their implementation, generate a new event to notify interested entities in the system. There is no one-to-one correspondence between a sensor or effector and a single fact. Some sensors will provide many facts, and others may only provide one. Some effectors may accept only a single fact; others may require many as inputs. Sensors and effectors have sensor or effector descriptions. When a sensor or effector is created, the creator must place its description in the knowledgebase, so other components can locate and use it. When a sensor is created, by the focus, or a subordinate part, an `AutonomicSensorDescription` is created. This description includes the sensor's name, whether the sensor is active or passive, any required input facts for the sensor and the set of possible Autonomic facts that the sensor generates. Sensors and effectors are usually referenced in the system policy files definition [14].

3.3.4 Policy Files

Our architecture model depends on the Autonomic Manager Toolset (AMTS) policy engine, which is provided by IBM as part of the Emerging Technologies Toolkit for policy evaluation and execution. Accordingly we are using the standard policy file format and structure used by the AMTS policy engine since we are actually creating an instance of the policy engine for each newly added system. We did some modifications to the way the engine handles the policies, in order to overcome some of the problems, which are not solved in the current engine release in addition to extending some of the engine functionalities. The AMTS policy engine is built on top of ABLE toolset, which is also provided by IBM [3]. The AMTS (Autonomic Manager Toolset) policy framework uses a Data Logic and Control (DLC) design model. This model specifies that an application consists of three major conceptual entities: the Data (input/sensor and output/effector data information, and the mechanism to create/obtain such data information), Logic (rules expressed as policies) and Control (triggering of rules, ruleset flow logic, etc). This design model requires that each of these components be separated. The Data contains only application data; it contains no logic nor any control or flow information. The Logic component contains only the Logic that uses the Data as input/output information; it has no knowledge of what the data will be and where to obtain them. Furthermore, it does not know how each individual policy (or rulesets) should be connected and when to trigger the policies (rulesets). The Control entity is responsible for the high-level application logic that links a series of logic modules (in the form of policies) together to form an application, and for the triggering mechanism to enable and invoke a sequence of policies. The design pattern with the AMTS policy framework supports the clean separation of data and logic. This design pattern promotes code reuse and simplicity; it also encourages the reuse and modularity of logic expressed as policies [14]. When an application developer uses the AMTS policy framework and programming model to create an application, the emphasis is on externalization of logic at various points in the application where logic and data can be cleanly separated. The design of the policy part of the autonomic manager toolkit is mainly based on the following model [14]:

- A policy in XML, which is based on the AMTS Policy schema (AMTSPolicy.xsd) and, together with the associated resource-mapping file (also

in XML based on the extension of AMTS Policy Schema (AMTSPolicyExtension.xsd) is created either by an authoring tool or GUI. The Current version of AMTS does not provide such tools. Both the AMTS policy schema (AMTSPolicy.xsd) and its extension (AMTSPolicyExtension.xsd) are available in the AMTS package: “com.ibm.autonomic.policy.shema”.

- In AMTS, the policy framework uses a pluggable architecture. The components needed to create an instance of policy execution are separated, developed separately and reused.

In general, to create a policy instance, the following components are needed [14]:

- A file or Java object (Document etc.) to represent the policy.
- A file or Java object (Document etc.) to represent resources (description of data) used in executing the policy.
- An underlying execution mechanism to execute the policy (a rule engine, or native Java).
- A translator, which translates the policy and mapping information into executable code appropriate for the underlying execution mechanism.
- An optional Knowledge Base (KB) where data can be retrieved, either through event subscription or through direct access to the Knowledge Base.

In AMTS, there are two ways in which a policy can be used [14]:

- A Consultative Policy is used when an application at a decision point directly consults a policy to obtain information or generate actions with some input data. The control of when to trigger the policy execution is solely the responsibility of the application itself. The input data can be provided in the form of direct inputs from the application or through the input Knowledge Base.
- An Event Driven Policy is used when an application at a decision point allows external events to trigger a policy to obtain information or generate actions based on the information provided by the triggering events. The Event Driven Policy instance must subscribe to the Knowledge Base for the specific information it wishes to receive.

In a typical application, it is possible to have multiple instances, and both types (Consultative and Event Driven) of policy usage working together. In subsequent versions of the AMTS, the ability to trigger an episode of policy execution by another policy will be available. Based on the current AMTSPolicy schema, a policy document (xml) includes [13-14]:

- A main Policy Document Element where information such as version and discipline are specified.
- A Condition Section which contains all condition definitions used in the policy document.
- An Action Section which contains all action definitions used in the policy document.
- A Policy Section which contains all the policy rules used in the policy document.
- At least one Policy Group which contains at least one policy rule.

The current AMTS policy grammar is based on an early draft of a standard AC (Autonomic Computing) policy grammar being developed by Heiko Ludwig and the Autonomic Computing Policy team. They intend to convert the current AMTS policy language to the standard as one becomes available. It is important to note the transient nature of the current AMTS policy grammar, and changes are likely in future AMTS releases [13-14].

3.3.5 Knowledge Base

The knowledge base acts to provide an isolation layer between knowledge creators and knowledge consumers. It exposes two sets of interfaces, one aimed at components, which provide knowledge, and one at those which consume knowledge. This permits users of knowledge stored in the knowledge base to access it uniformly, while the knowledge can be provided transparently from a broad set of components. The knowledge base acts as a publish/subscribe service, permitting components within an autonomic manager to subscribe to changes in knowledge held within the knowledge base. The knowledge base will provide basic support for structuring knowledge in terms of ontology, specialization and aggregation. [13,18]

Figure 3.8 presents a snapshot for the condition section of a sample policy file whereas figure 3.9 presents a snapshot for the action section. Both figures are shown again in the experimental work chapter (chapter 4) since they are used in one of the experiments. The AMTS [13-15] documentation contains the detailed description of the AMTS rules and mechanism. The reader may refer to it for more details.

Resource Optimization Property Manager For Autonomic Computing

```
<?xml version="1.0" encoding="UTF-8" ?>
- <PerformanceMonitorPolicyDocument version="1.0" discipline="PerformanceMonitorPolicy">
  <!-- ... -->
  <!-- ... -->
  - <ConditionSection>
    - <SimpleCondition policyElementId="c1">
      <Operator>greaterThan</Operator>
      - <Operand1>
        - <Expression>
          <VariableName>measuredValue</VariableName>
        </Expression>
      </Operand1>
      - <Operand2>
        - <Expression>
          <VariableName>expectedResponseTime</VariableName>
        </Expression>
      </Operand2>
    </SimpleCondition>
  </ConditionSection>
+ <ActionSection>
+ <ACPolicySection>
- <ACPolicyGroup policyElementId="g1">
  <ACPolicyRef name="p1" />
</ACPolicyGroup>
</PerformanceMonitorPolicyDocument>
```

Figure 3.8: Web Policy Precondition Section

```
+ <ConditionSection>
- <ActionSection>
  - <SimplePolicyAction policyElementId="a1">
    - <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
      - <Parameter>
        - <Expression>
          <Literal>"Policy"</Literal>
        </Expression>
        - <Expression>
          <Literal>"HTTP"</Literal>
        </Expression>
      </Parameter>
    </MethodExpression>
  </SimplePolicyAction>
</ActionSection>
- <ACPolicySection>
  - <ACPolicy policyElementId="p1">
    - <Precondition>
      <SimpleConditionRef>c1</SimpleConditionRef>
    </Precondition>
    <SimplePolicyActionRef>a1</SimplePolicyActionRef>
  </ACPolicy>
</ACPolicySection>
- <ACPolicyGroup policyElementId="g1">
  <ACPolicyRef name="p1" />
</ACPolicyGroup>
</PerformanceMonitorPolicyDocument>
```

Figure 3.9: Web Policy Action Section

3.3.6 Extension Policy File

The current version of the AMTS policy engine included in ETTK 1.1 [15] does not support all the mentioned features in the toolset manual. Many of the important features such as the ability to trigger an episode of policy execution by another policy is not implemented yet. Another missing feature is the ability to call a sensor function inside a consultative policy, as this is only allowed in polling sensors, which are defined in event driven policies only. The current version does not also support the parameterized sensor function call for any sensor type. That means functions that take no parameters are the only type of functions allowed to be attached to a sensor, which is definitely not a practical manner. Additionally, in the current version the sensors and effectors that are defined in the XML files should be explicitly hardwired in the code during compile time, which is a procedure we want to avoid. Actually we want to completely separate the implementation from the given data, as the system administrator should not need to write or modify or recompile any piece of code order to register or monitor any of his systems. This is not the case in the current version. All the previously mentioned problems generated the need to create a mechanism to be used as a work around solution for these problems.

We finally arrived at what is labeled the policy extension file to be a practical solution for most of the above mentioned problems. We first explain the syntax of the file and then we will move to the semantic description. We had to set our own file structure in order to guarantee the most flexible and practical form. We divided the extension file into two sections. The Sensors section, which starts with the “Sensors:” keyword and an effector section, which starts with the “ Effectors:” keyword. We used the “#” sign to precede any comments in the file. The file can include as many empty lines and spaces as necessary since the parser skips all the empty lines and spaces. The new line sequence character however separates any declaration of a sensor or effector within the file (i.e. one declaration per line). The single declaration can span more than one line provided that they are not separated by the new line sequence. All declared parameters are either declared as of type value or method. Inside the condition section of the AMTS policy file definition, the creator can use any variable name on both sides of the operator. For example you can write “ IF MeasureValue > 10” or you can write “IF MeasuredValue > ThresholdValue”. Actually the

engine assumes that the two values are defined within the knowledgebase; otherwise it will throw an exception during policy evaluation. The AMTS policy engine does not provide any tool to define the associated values with each declared variable in the policy dynamically. Unfortunately the only possible way to achieve this is to explicitly add those variables into the engine KB within the code and then compile it. So the work around that we have figured out for this problem is to declare the initial value of all the variables used inside the policy files in the policy extension file. Hence, the user declares the previously mentioned variables as follow:

- “*value;MeasuredValue;10;*”
- “*value; ThresholdValue;100;*”

What actually happens is that the parser parses the extension file of each defined policy before the main policy file is uploaded into the engine, and it detects all the declared value parameters then starts to add them up into the knowledgebase. Thus they can be referenced by any of their corresponding policies when needed. This mechanism helped in externalizing the variable declaration, which was not possible in the current AMTS policy version. The second type of variable declaration is the method. As mentioned earlier, function based sensors could only be defined as polling sensor in the AMTS policy engine and they can only be associated with non-parameterized functions. This means that the sensor function cannot accept any parameters and can only be defined in event driven policies. Alternatively we have provided the capability of defining parameterized function calls inside consultative policies by declaring them in the corresponding extension policy file. Actually sensors that are going to be associated to a function call have to be declared twice inside the policy extension file. The first time to indicate its default value inside the KB and the second time is to declare the associated function call. Each defined parameter value and its associated function call is stored in a hash table, and whenever the policy is consulted our policy server module goes through all the defined sensors of that policy and starts calling their associated functions to update their values in the KB. An example for a declaration of the sensor called “MeasuredValue”, which is associated to a function call, is shown below. As noted, we first mention that this sensor declaration is of type method association sensor. Then we mention the address of the machine to which this function call should go (e.g. the remote machine name shown in the example is “TestX”). Then we mention the name of the

sensor “MeasuredValue” then the function return type “int” preceded by the package name and class name “AutonomicPackage.ProbingStation” which contains this function. Then the function name itself “GetServStatus”. Finally we mention the entire parameter list for the function itself each parameter separated by a “,”.

- *method;TestX;MeasuredValue;int|AutonomicPackage.ProbingStation|GetServStatus|localhost,/manual/misc/FAQ.html#name,cls,false,;*

The above shown example presented a call to an external function associated with a certain sensor. What we mean by an external function is that the intended function or batch resides in a remote machine. Additionally, as will be explained later, a sensor could be associated to a local function call. The next example shown below presents the syntax used to call a built-in function called “GetCpuUsage” which retrieves the current CPU usage of the intended machine. What is meant by a built-in function is the set of functions that are provided by the property manager as ready-made functions that the user can make use of in his policies without providing an implementation for. Each property manager will have his own set of built-in functions and that should facilitate the process of building useful policies by the system administrator. In our prototype we have implemented the “GetCpuUsage” function, which retrieves the current CPU utilization reading of a running machine or of a certain process. The sensor shown below is called “CPU_USAGE” and is associated with the built-in function ”GetCpuUsage” which takes two parameters the first one indicates the delay between each reading iteration and the second indicates the number of iterations that should be averaged and returned back to the user as an average CPU utilization. Hence, the function returns the average of five readings each separated by a one second reading. The class name of this function is called “PerfCounter” which contains other useful functions such as memory and disk space readings. Again the address of the machine in which the function will run is called “TextX”.

- *method;TestX;CPU_USAGE;BuiltIn|PerfCounter|GetCpuUsage|1000,5,;*

As promised by IBM the upcoming versions of the AMTS policy engine will support parameterized function calls in all type of policies. So in the near future we should be able to

give up some of the above mentioned workarounds as they will already be implemented (Please refer to Appendix C for a list of all the implemented workarounds). The last option that we have provided through the policy extension file, is the ability to identify a sensor which is actually a pointer to another sensor defined in a different policy. The example given below is a sensor definition that is included inside the security policy file. It defines a sensor called “MeasuredValue” which is actually a “sensor listener “ associated to a sensor with the same name defined in the policy called “WebSiteP”. Then we assign a priority to that sensor to indicate the order in which all the registered polices will get the notification about the sensor value change. So if the sensor is assigned a certain priority in the security policy which is higher than the one assigned to it in the Website policy, the security policy will always be notified about the sensor value change before the website policy. Throughout this mechanism we provide a more efficient approach to the sensors definitions and reusability in more than one policy.

- *Notify;measuredValue;SensorListener,WebSiteP,7,;*

For the effector section things are much simpler than they are in the sensor section as the current AMTS policy version supports the call of a parameterized function effector. Fortunately this has relieved a great amount of work which could have been similar to the one achieved in the sensor part. The only problem concerning the effector part is that it should also be defined explicitly in the code to be recognized by the policy engine whenever the action section is executed. To overcome this problem we defined a default effector function called “sendAlertEffector”. This function accepts two parameters. The first parameter specifies the call type. The supported types are either a function call or a policy call, which is also not supported by the current AMTS version. The second parameter either specifies the function name and its calling parameters in case it is a function call or the policy name in case it is a policy call. The example shown below is for effector declaration syntax inside the policy extension file. Definitely the effector function could be named anything other than “SendAlertEffector” provided that this name is used inside the AMTS policy files. Figure 3.10 provides a snapshot for a function call effector in the policy action section, which makes a call to a function named “AllocateResrouce”, whereas Figure 3.11 provides a policy call effector, which makes a call to a policy named “HTTP”.

- *Method;TestX;sendAlertEffector;*

```
- <SimplePolicyAction policyElementId="a2">
- <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
- <Parameter>
- <Expression>
  <Literal>"FunctionCall"</Literal>
  </Expression>
- <Expression>
  <Literal>"BuiltIn | ResourceAllocator | AllocateResource | CPU,HTTPServer,true,|;"</Literal>
  </Expression>
</Parameter>
</MethodExpression>
</SimplePolicyAction>
```

Figure 3.10: Effector Function Call Snapshot

```
- <ActionSection>
- <SimplePolicyAction policyElementId="a1">
- <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
- <Parameter>
- <Expression>
  <!--
  <Literal>"Policy"</Literal>
  </Expression>
  >
- <Expression>
  <Literal>"HTTP"</Literal>
  </Expression>
</Parameter>
</MethodExpression>
</SimplePolicyAction>
</ActionSection>
```

Figure 3.11: Effector Policy Call Snapshot

3.3.7 Event Notification

The AMTS toolset provides an event notification mechanism that would have been very useful if it was fully implemented. Unfortunately some parts of this mechanism are not yet supported in the current version (ETTK 1.1) however we were able to make the best use of the implemented part and to implement some work around to simulate some of the missing parts. The toolset is largely event-driven. An element's objects spend most of their existence waiting for events to happen. When the element handles a request, events begin flowing through the components. Event flows fall into two categories, synchronous and asynchronous. Many of the event flows are synchronous, with a request leading to a response. Some internal flows are asynchronous, often when one component sends a notification to another component as part of its processing. The manager generates many of the events that flow through it. Timer-driven sensors actively poll the environment, and when they detect changes they drive events into other portions of the element. Some policies require the active, regular evaluation of the element's environment, and the regular evaluation of the policies. Timer events trigger these policies [14].

3.3.7.1 Event subscription

Components in an autonomic manager request notification of relevant events by subscribing to the components that generate these events. When component A wants to be notified when another component B changes state, it creates an event subscription on component B. Component A passes to component B the request for events, with itself as the event listener. As long as the subscription is in place, B will call A whenever B changes state. Event subscription and delivery is a structured process. Components use two forms of event subscription. The first is used by components that want to know an event has occurred but do not take direct action based on the event. This form is used by components such as loggers, monitors and debuggers. The second form of event subscription is used by components that take actions based on events. These subscriptions are ordered by priority, and called in priority order. When the event listener completes its processing, it can return one of three possible results to the notifying component. It can return the event, unchanged, indicating that the event listener should continue notifying lower priority listeners. It can return a

modified version of the event, which the event notifier will pass on to the lower priority listeners. It can return a null event, indicating that it has fully processed the event, and no lower priority listeners should be notified of the event [14].

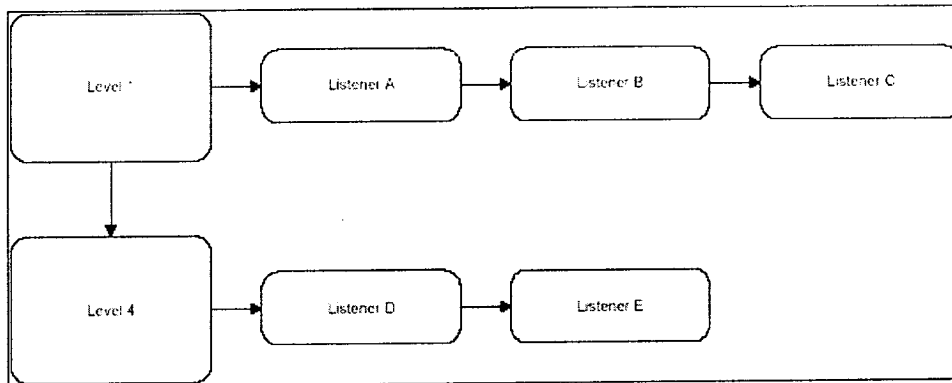


Figure 3.12: Event Delivery Chains [14]

3.3.7.2 Delivery Chain example

In Figure 3.12 we show five event listeners at two separate priority levels, where level 1 has a higher priority than level 4 in receiving the incoming events. The event will be propagated first to listener A, then B, then C, followed by D and E. Each listener may perform exactly three actions. They may consume the event, propagate the event or annotate and propagate the event. For example, Listener A could propagate the event to listener B, which would annotate it with additional information. Listener C would receive the annotated event, and could further annotate it. Listener D would receive the twice-annotated event, and could consume it, thereby terminating the processing for the event, in which case, listener E will never receive any version of the event [14].

3.3.7.3 Priorities

The AMTS uses an extension of the normal java event listening model. The AMTS extends the model in two ways. The model introduces priorities to event delivery. In addition, it permits event listeners to influence what lower priority listeners receive, including the prevention of the delivery of the event to lower priority listeners. Event

delivery is done in priority order, from 1 to the maximum value held in an integer. Within a priority level, there is no guarantee of delivery in a specified order. So, in figure 3.12, Listeners A, B and C, having registered at priority 1, are ensured they will be called before any listeners at lower priorities (D and E) but they can make no assumptions about which of them will be called first [14].

3.3.7.4 Missing features

The current version of the AMTS policy engine can only return the event, unchanged, indicating that the event listener should continue notifying lower priority listeners when it completes its processing. Event modification or complete event consumption is not yet supported in the current version. In our architecture we are making use of prioritized event notification mechanism to be used among the different property managers and the different policies. As more than one property manager might be interested in the same event notification, each according to its priority, the concerned property manager should either consume that event after handling it or pass it over to the next property manager in case it is not interested in it. For example, if the security property manager and the optimization property manager are both concerned with the website response time value change, consequently both of them will register to the same related event notification with a different priority. The security manager will have a higher priority since it is more important to verify the security concerns before starting an optimization investigation. So the security manager will be the first to be notified about the response time value change. We will assume that it has discovered a security problem with the web site and it was able to handle it. So the security manager should have consumed the event notification, as it is not necessary any more to pass it to the optimization manager. Unfortunately this scenario is not supported in the current version so the event will always be passed to the optimization manager, which creates a sharing violation, as the optimization manager will start to handle the event at the same time that the security manager might still be working on the problem. In order to temporarily over come this issue, until the full solution is hopefully implemented in the coming version of the AMTS engine, we have created what is known as Global Locks. Global Locks are similar to the semaphores, and monitors multithreading OS solutions. The

first property manager receives a shared event notification, sets the lock on to indicate that it is working on the problem. Once it is done it releases the lock so that other property managers can proceed with handling the event. This work around was implemented in the resource optimization property manager and presented in Experiment 4 in the experimental section of our work Chapter 4). This work around has provided a solution to the event notification sharing violation, which might lead to more complex problems during policy handling. As we mentioned above once the AMTS new release is ready, we will abandon this work around.

3.3.8 Messaging Using JMS.

As mentioned earlier our architecture is client-server based architecture, however in they do not communicate directly to each other. There is an intermediate communication layer, which adds more robustness, reliability, and availability to the communication and that is achieved throughout the use of the messaging service known as Java Messaging Service (JMS). Actually the JMS is a standard by itself but more than one vendor has provided an implementation for the JMS standard. Specifically we have used the JMS server provided by Sun, which is included in the J2EE package. The JMS server APIs provided by sun are easy to use and to hook up applications to. We will give a brief description for the java messaging service concepts and benefits. Additionally the reader can refer back to [29] for more details.

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages. Messaging enables distributed communication, which is loosely coupled. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know any-thing about the sender. The sender and the receiver need to know only what message format and what destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods. Messaging also differs from electronic mail (e-mail), which is a method of communication between people or between software

applications and people. Messaging is used for communication between software applications or software components [30].

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics which allow programs written in the Java programming language to communicate with other messaging implementations. The JMS API minimizes the set of concepts that a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain. The JMS API enables communication, which is not only loosely coupled but also [31]:

- Asynchronous. A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
- Reliable. The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

In our architecture we use the point to point messaging provided by the JMS server rather than the Publish/Subscribe Messaging since it is more suitable to our intended design as will shortly be explained. A point-to-point (PTP) product or application is built around the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and the receiving clients extract messages from the queue(s) established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire. PTP messaging has the following characteristics and is illustrated in Figure 3.13.

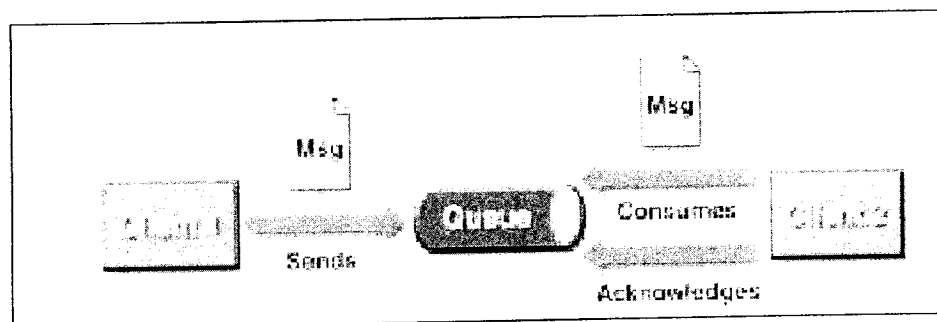


Figure 3.13: PTP Messaging [31]

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message. PTP messaging is usually used when one consumer must process every sent message successfully.

In order to facilitate the communication between the server side and the different clients, the property manager creates one messaging queue per client machine named after the machine address. This means that if more than one managed application is running on the same client machine, they will all share the same messaging queue. Also if more than one property manager is communicating with the same client machine, they will also use the same messaging queue named after the machine address. Thus the messaging queue is created per hosting machine not per running application. Figure 3.14 depicts an overview for the property manager overall architecture. The entire architecture model was designed and implemented within this research work and the AMTS policy engine was integrated into it. On the server side the Command Dispatcher/Receiver module takes care of sending all the outgoing property manager messages to the different clients, and it also handles all the incoming messages and redirects them to the caller. All the JMS messages hold timestamps and extra flags that enable the sender to specify the receiver and consequently the receiver to recognize the message sender. On the client side the command receiver/executer module takes care of all the incoming messages and it routes them accordingly, collects the result back, and finally sends them back to the appropriate JMS queue. This mechanism extremely smooths the communication between the client and the server side, and it increases the flexibility of adding or removing clients transparently without affecting the server side, which makes the client/server sides loosely coupled.

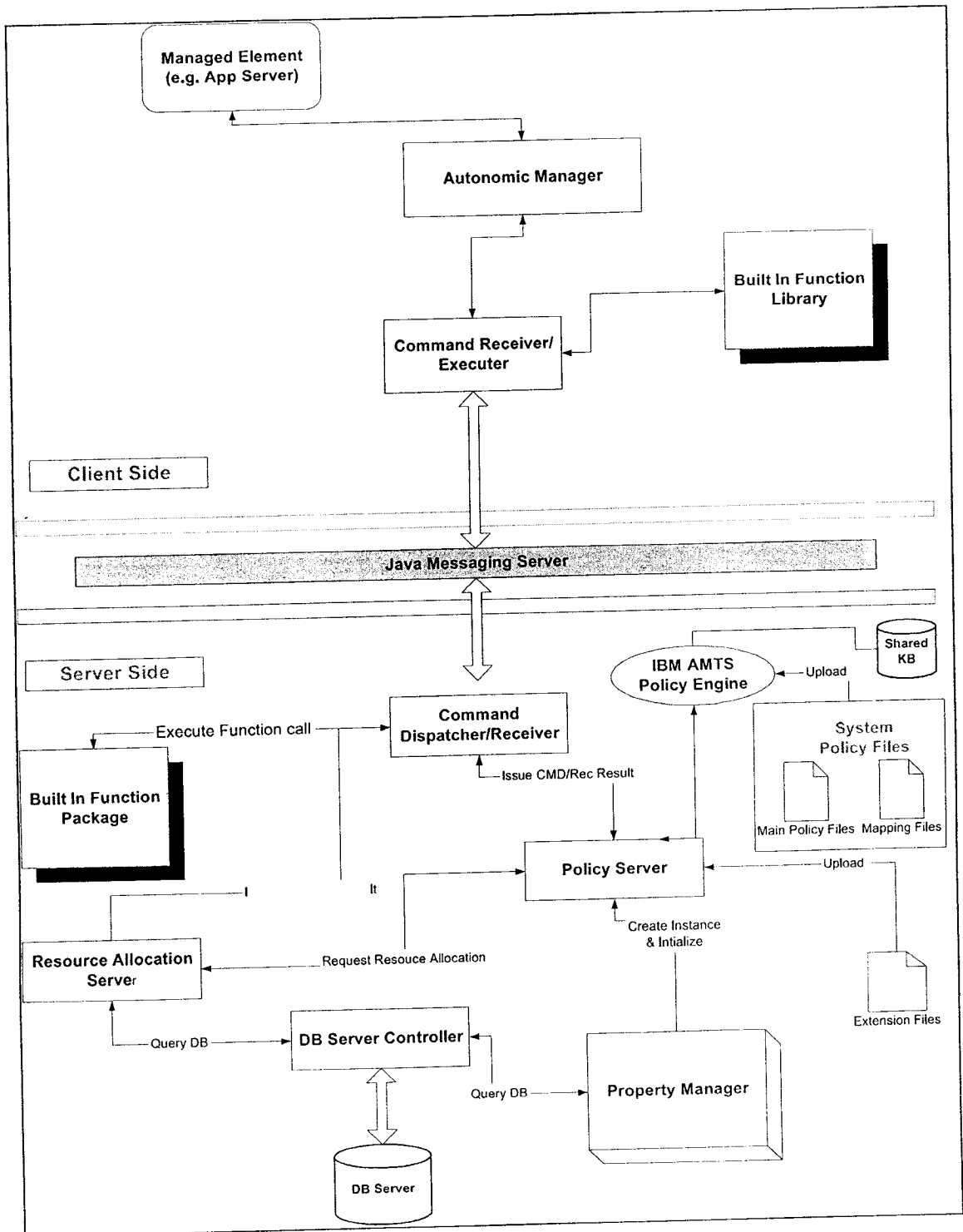


Figure 3.14: Property Manager Overall Architecture

3.4 The Server Side.

Figure 3.14 presents a general overview for the different components, which together formulates the structure of the autonomic property manager. It illustrates the components that exist on both the server and client sides in addition to the communication medium (JMS server) through which the two parties communicate. Figure 3.15 presents a class diagram for all the classes that build up the autonomic property manager and the packages that are used by the member classes. All the shown classes were implemented within this research work except for the ready-made packages that are provided by the integrated technologies. The “J2EE JMS package” [21] includes all the important JMS server APIs which both the “*CommandDispatcher*” and “*CommandReceiver*” classes use to communicate with the JMS server. The “AMTS Policy Engine Package” is the package that contains all the AMTS policy engine APIs used by the “*PolicyServer*” class to build up the property manager policies. The last package is the “PropertyManagerPack” which contains all the property manager specific functions such as the “GetCPU” function that is used to retrieve the CPU utilization of a certain machine or process. It can also contain any additional functions that are property manager specific. Any call to one of the built-in functions provided by the property manager (i.e. included in the package), is resolved dynamically from the used function name in the corresponding policy file during runtime. For example, the administrator can use the built function “GetCPU” in any of the managed systems policy files as a sensor name. Actually, this is where the strength of this model stands. This mechanism means that we can add a set of new functionalities provided by a new property manager, by simply adding the new classes into that package directory. Or we can even replace one of the old functionalities by replacing the old class with the new one, which contains the new functionality. Hence, there is no need to recompile or generate any code parts in order to support new functionalities in the system. The class diagram shown in Figure 3.15 does not show the attributes and member functions of each class due to the space limitation to include all the details in one diagram. The detailed diagram of each class is included in the Appendix D. Initially we explain how these classes communicate together and the role of each class. Upon the very first time of the system start up, the “Global Manager” class creates an instance of the “Property Manager” class for each property manager. It then calls the “Start

system can use its own KB instance. All our experimental work was conducted using one shared KB instance, since there was no need to use more than one instance. The sequence diagram shown in Figure 3.16 depicts the above mentioned property manager initialization and startup sequence.

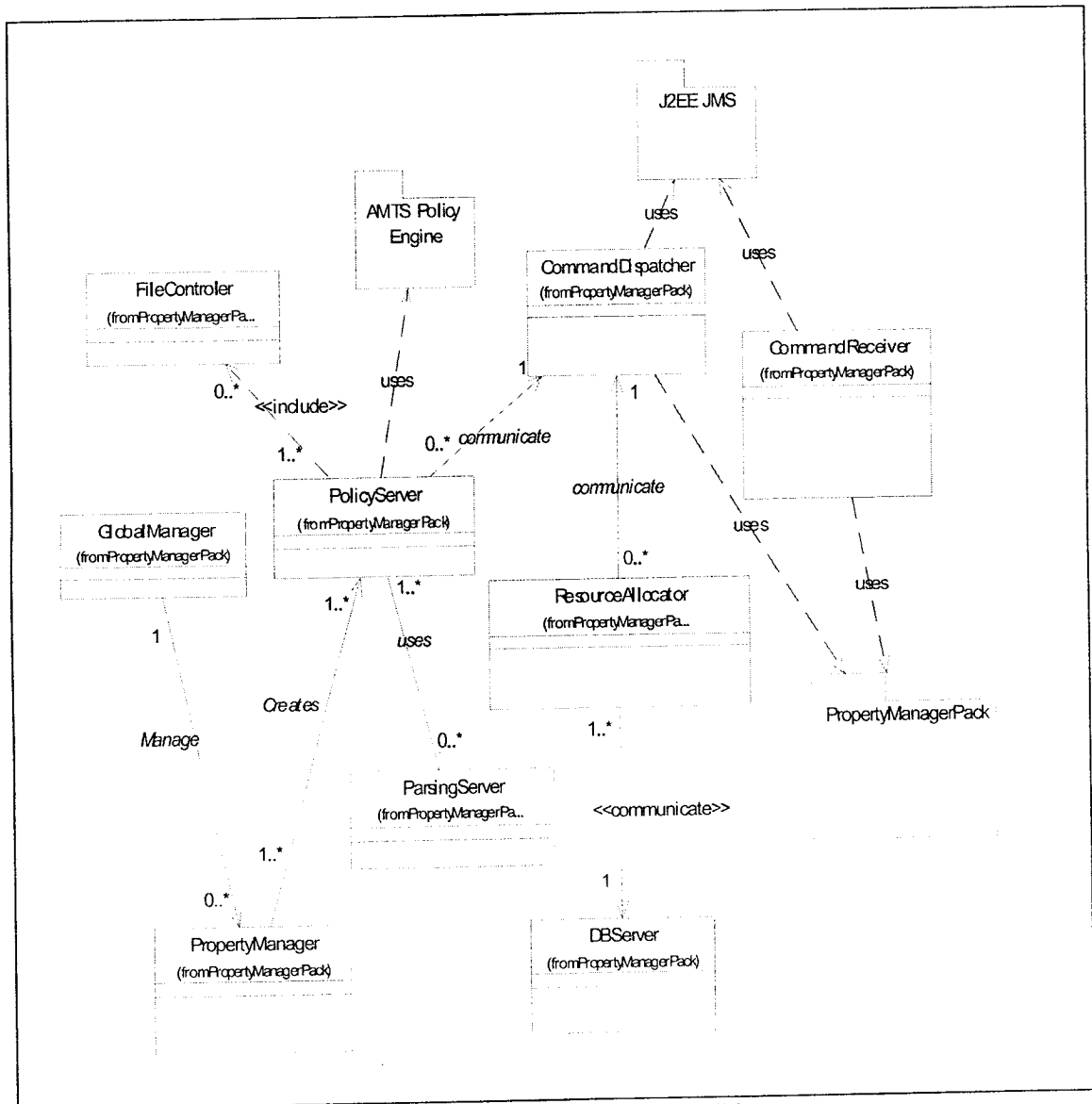


Figure 3.15: Property Manager Class Diagram

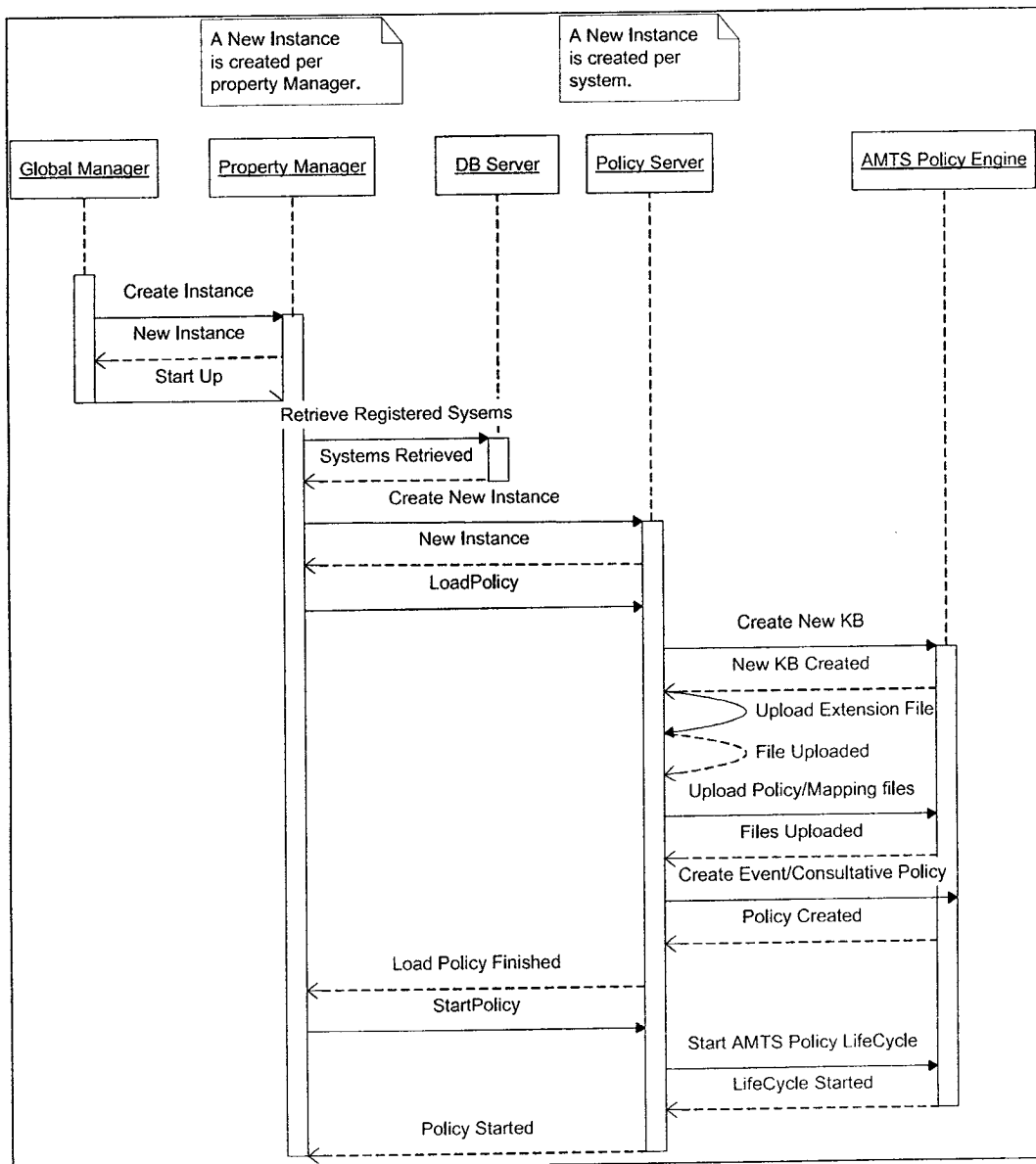


Figure 3.16: Property Manager Overall Architecture

3.4.1 Command Dispatcher

In this section we explain the mechanism by which the “policy server” class communicates with the command dispatcher once it triggers a sensor or an effector call as a result of a policy evaluation. Initially when a sensor or an effector call is invoked, the policy server calls the member function “CommandClassifier” which takes two parameters. These parameters are the sensor or effector name, and the type of the call (i.e. is it an effector or sensor). If it is an effector, it sends the call right away to the command dispatcher since all the needed information is already included in the call itself (remember the current AMTS version support parameterized function calls for effectors only). On the other hand if it is a sensor call, this function will retrieve the full parameterized function call associated with the supplied sensor name included in the extension policy file. Then it sends the full information to the command dispatcher. The command dispatcher verifies whether this is a local call or a remote call. If it is a local call it does the execution, collects the result back, and sends it to the caller. In case it is a remote call, it dispatches it to the corresponding JMS queue and collects the result back for sending it to the caller. The Command Dispatcher State Diagram shown in Figure 3.17 illustrates the previously mentioned scenario.

3.4.2 Resource Allocation

The resource allocation class is one of the major modules that formulate the resource optimization manager. Any resource request call, which is embedded in one of the systems policy files are redirected and handled by this class. This class follows a certain mechanism that will be explained shortly in order to allocate some of the requested resources if possible. It does not guarantee that it will always be able to allocate the requested resource since the final decision is actually made by the autonomic manager managing the resource itself. However it does some preprocessing in order to allocate the best candidate system that can give up some of the requested resource at the request time. As the Resource Allocation server class receives a request from a certain system, which requires additional allocation for a

specific resource (the requester system name and required resource type are provided as parameters), it does the following:

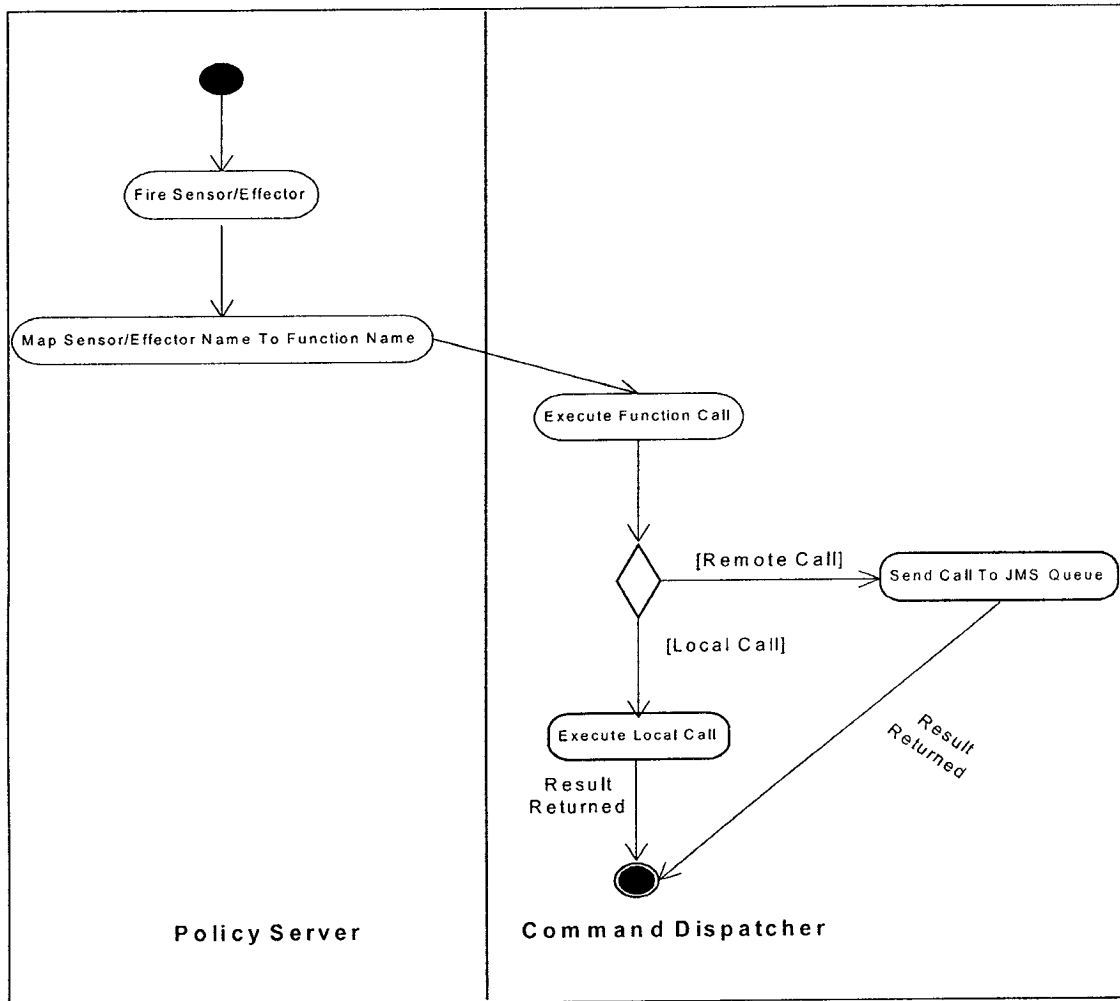


Figure 3.17: Command Dispatcher Activity Diagram

- It runs a query that looks for all the systems that are currently using the same resource, operating out of their peak hours, and have lower priority than the requesting system.
- It loops on these systems to update their resource counter readings by calling the appropriate function associated with each system to ensure that the current readings reflect the latest counter updates.

- The systems are sorted according to their priority and updated resource consumption.
- It starts to call the first system resource acquisition defined function, which informs the managed resource about the resource acquisition request.
- If a system returns a Zero it means that it accepts the donation and consequently the resource is released; otherwise the system continues looping on the rest of the systems. If this function succeeds in finding some resource it will return a (zero) otherwise it will return (-1) to indicate a failure.

The previously explained sequence is illustrated through the sequence diagram shown in Figure 3.19. We recall here the ER diagram in Figure 3.6, which shows the “AUT_ResourceConsumption” table in which all the resource acquisition functions, and counter readings of each system are stored. Other useful information is also stored in the table such as the average resource consumption for the system, the current resource consumption, and the last update to the value reflected in the current resource consumption. All resource allocation transactions, which succeed or fail are stored in the table called “AUT_ResourceTransaction” where the requesting system name, donor system name, requested resource, and transaction time are all stored. This data could be used for statistical and analysis reporting and for graph generation. It could also be used for transaction rollback when needed or transaction forward look up tables, which can help in allocating resources in the future, but these features are not supported in the current version. However they will be useful for future enhancements. Finally there is a view called “V_AUT_PeakTimes “ which is created in the database server to constantly hold all the systems that are working on their peak times to be excluded from the resource-giving list during resource allocation system search. Figure 3.18 displays a snapshot for the previously mentioned view.

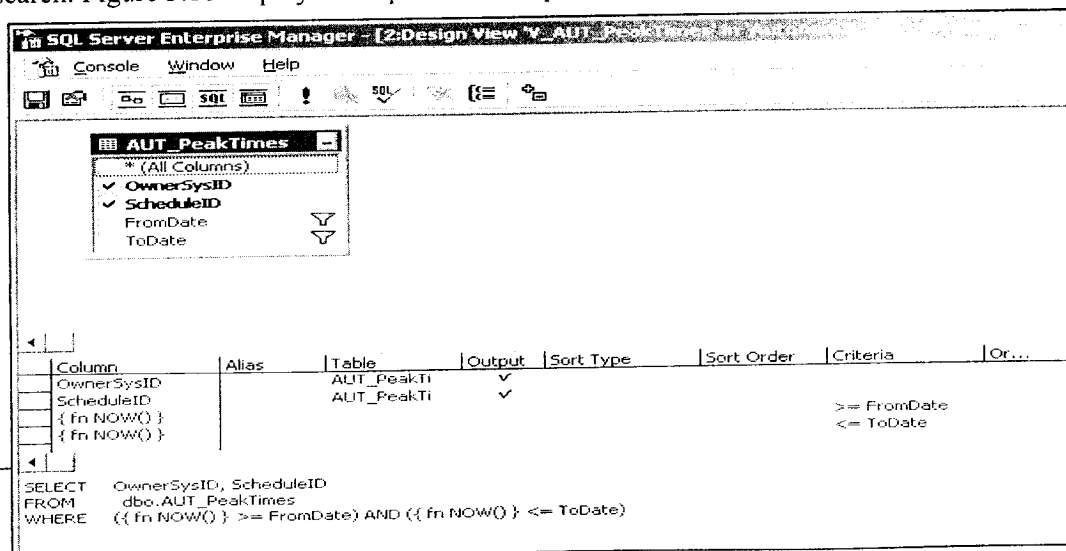


Figure 3.18: Off Peak Time view

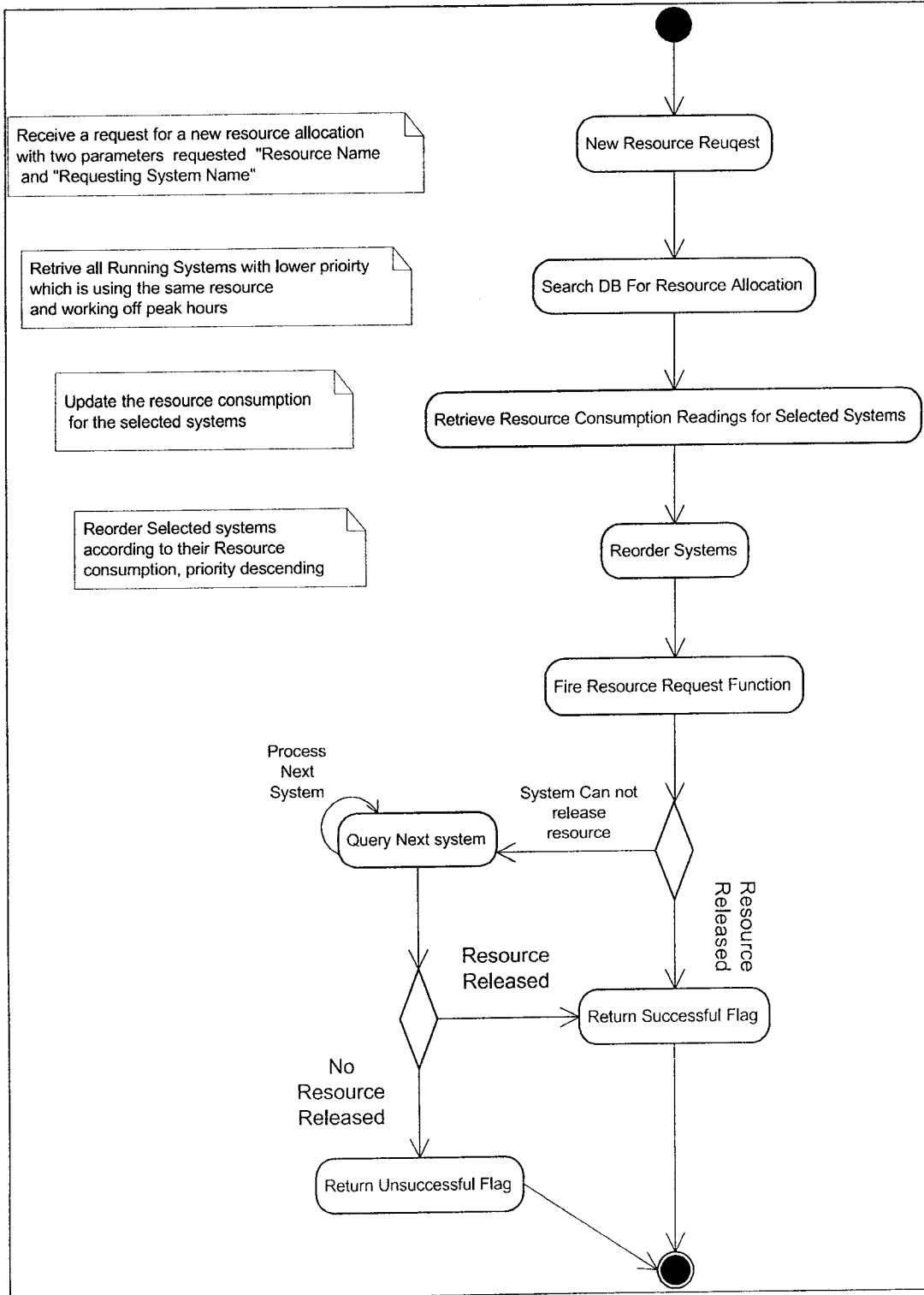


Figure 3.19: Resource Allocation Activity Diagram

3.5 The Client Side

Finally we come to the description of the client side components or classes. As was shown in figure 3.14 there is a command receiver/executer class, which takes care of receiving all the incoming messages that are related to the machine in which it resides. Once it receives the message it does a filtration process in order to check if the coming message holds a built in command that is supported by the property manager package or is an external call to a local class or batch. It executes the call accordingly and then returns the result back through the JMS server. As we had mentioned earlier the receiver supports the call to any class, function, and system batch or script that are contained in the hosting machine as long as the full and correct path has been provided in the message call. The command receiver state diagram is shown in figure 3.20

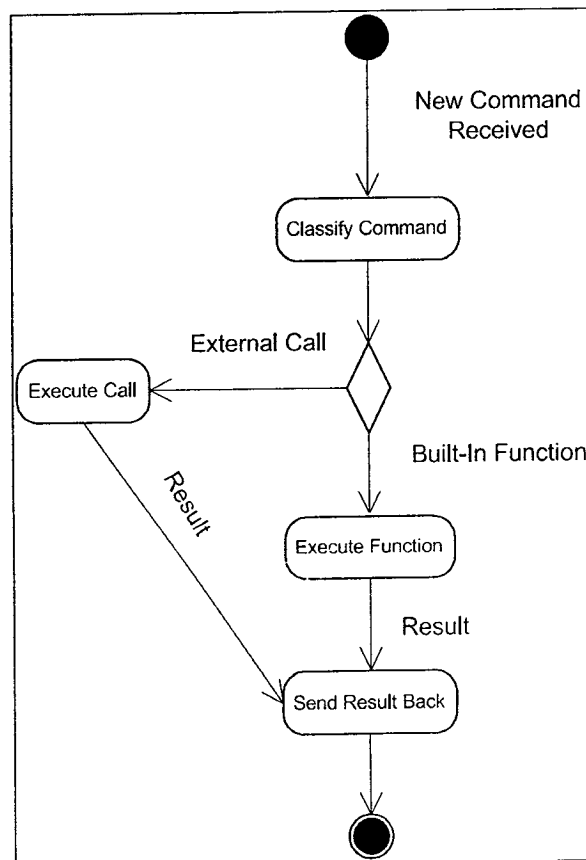


Figure 3.20: Command Receiver Activity Diagram

3.6 Resource Optimization Property Manager Implementation.

According to IBM's definition of the Self Optimization property: "Autonomic systems will continually seek ways to improve their operation, identifying and seizing opportunities to make themselves more efficient in performance or cost [1]". Within the scope of this research, this statement is interpreted to mean that the Resource Optimization Manager enhances the performance of the managed systems according to a set of predefined policies which control its behavior in addition to the resource management functionality. This ultimately leads to the optimization of the managed systems according to the policies supplied by the systems administrators. Hence, it is important to clarify here that our proposed Resource Optimization Manager is built on a rule-based optimization strategy rather than a resource optimization function.

Since it will be difficult (in terms of the time frame and available resources) to design and implement each of the four basic autonomic property managers which together form the Self Management autonomic property (Configuration Manager, Self Healing Manager, Security Manager, Optimization Manger), we implemented only a prototype for the resource optimization property manager as a proof of concept for the efficiency of our previously proposed architecture. In reality the resource optimization manager should be communicating with an autonomic manager (at level 1), which is controlling a managed resource such as an HTTP or Application server. However since we were not able to get any of the already existing autonomic managers implementation such as the AutoTune agent [41], we were directly communicating with the managed element in our implemented model. We developed a set of classes and batches through which we can communicate and invoke some of the HTTP apache server [5] and WebShpere application server [37-28] functionalities. For example, we developed a class, which provides an interface for the apache HTTP server and through which we can retrieve or set the value of any tuning parameters in addition to the ability to stop or restart the apache server. We also developed batches that can stop or restart a selected web application, which is hosted by the WebSphere applicationserver. All of the implemented tools were efficiently used throughout the experimental work to provide a simulated communication mechanism between the autonomic property manager and the autonomic manager.

3.7 Used Technologies.

3.7.1 Emerging Technology Toolkit (ETTK)

With regards to the programming language, tool kits, and technologies that were used in the development and prototyping process, we decided to use the Java programming language and some of the emerging technology tools provided by IBM particularly for the autonomic managers development. We chose the Java language, for its platform independence, which provides more flexibility in terms of both implementation and mobility feature in addition to the other known benefits of the java programming language as an object oriented programming language. We also used the Autonomic Manager Tool Set (AMTS), which is part of the Emerging Technologies Toolkit (ETTK) [15] provided by IBM. The ETTK provides a run-time environment, as well as demos, examples, and additional tools to design and showcase emerging technologies. The toolkit also provides introductory material for developers to easily get started with Autonomic Web Services and Grid-related applications. The Toolkit prototypes emerging technologies and allows developers to experiment with creating their own applications and demos [13]. The ETTK evolved from the package formerly known as the Web Services Toolkit (WSTK). With the renaming of the WSTK package to ETTK, the scope of technologies has been expanded. In addition to Web services, emerging autonomic and grid technologies are now being integrated into the package. With this new direction, the toolkit has also changed the way it is packaged (i.e. it is now composed of separate components or tracks which can be used separately). Related technologies are now grouped together into "tracks". The track that we are mainly concerned with is the Autonomic Manager Tool Set track. The Autonomic Manager toolset (AMTS) is designed to aid in the creation of the management portion of Autonomic Elements. The toolset can be used to build a wide variety of elements, ranging from simple components which monitor low level resources, to complete autonomic elements managing large complexes of distributed computing middleware. In this toolset a set of components are provided which can be used in a variety of ways to yield many different solutions. The toolset includes a "skeleton" which combines the toolset's components into one very

specific style of an autonomic element, but this style is not required, nor will it be appropriate for all autonomic elements. The toolset consists of several sorts of components. Some components are essentially standalone and orthogonal to the rest of the toolset [14]. Other components leverage each other more extensively. Each component can be replaced, or used in parallel with a similar component, but often at the expense of limiting the usability of other parts of the toolset. Whenever possible, interfaces are provided to permit complete substitution of components by other components with similar capabilities. The current release of the toolkit represents a first, incomplete effort at separating the toolset elements from the framework. Subsequent releases are expected to improve this separation. The toolset includes support for a preliminary form of a policy enforcement point, and an early experience version of a policy engine for evaluating policies. We had previously mentioned some of the AMTS fundamentals in section 3.3.4, as it is a core component of our architecture. If the reader wishes to obtain more detailed information about the AMTS component please refer to the ETTK [18] documentation.

3.7.2 Intended Platform And Users.

We used Windows 2000 Professional and Windows XP Professional as the intended design and testing platforms. IBM provides a windows OS version for all the needed toolkits as well as other platforms. However we prefer to use the windows platform for ease of use and familiarity of the OS structure. The users that our model is addressing are primarily system administrators, who can use the policy definition language to provide information about their systems and to register systems into the property manager that they select. In addition we address the programmers who need to define a model which they can use to build their own property managers for any new emerging property that they believe is important to be defined in an autonomic system. Many different business branches and IT industries will benefit from this model, such as data centers in large companies in providing a more robust system with the maximum resource utilization and minimum down times, by employing a more complete autonomic model.

3.8 Autonomic Property Manager Architecture Wrap-up

The intention of the provided architecture is to provide an efficient, flexible, and extensible model that will satisfy most of the autonomic system requirements and consequently provide a self-management-capable system. In our model a property manager will usually contain a set of default built in monitoring functions that are related to its specialization (the property that it represents). For example the resource optimization property manager will contain default functions that monitor the CPU usage, memory usage, and storage usage for a specific machine, system, or process and some other default monitor metrics, which fit the functionality of the concerned property manager. What we provide in our model is a flexible approach to enhance and extend the monitoring capability of any property manager. In other words, a means to define new monitoring functions, that are not built into the property manager and are supplied by an external party. These new monitoring functions will be in the form of a set of supplied classes, which could be instantiated from the property manager through the policy files (i.e. included as sensors and effectors), and hence be called to return the required inputs. Using an external tool or executable program, which could be called to return an input in a certain file format, provides other form of extensibility. For example, a log translator, a probing station program, etc.. In addition, each property manager will contain some fixed logic, which is related to the job it performs. For example the resource optimization property manager has the ability to look for extra resource allocation when a system is in need of extra resources. As previously illustrated, property managers are managed through a graphical management console through which the system administrator can administer and monitor all the registered systems effectively. For example, the system administrator will be able to use this console to change the priority of a certain system, or even stop or start a certain system. This way a group of systems can easily be administrated concurrently. Finally we conclude that that proposed architecture provides a model in which any system can fit and together they form an autonomic system which is capable of managing itself. The whole architecture was developed within this research work and the AMTS policy engine was only integrated as a ready-made component. A full prototype was implemented for the ROPM architecture. The same architecture can be used to provide the basis for building the architectures of any of the other property managers.

Chapter 4. Experimental Work and Analysis of Results.

4.1 Introduction

The purpose of this experimental work is to demonstrate the practicality of the proposed design work of the autonomic property manager notion. All the experiments described in this chapter were conducted using the prototype implementation of the Resource Optimization Property Manager (ROPM). In this chapter we present a complete analysis and description for each experimental test case as well as the full environment and parameter settings used during that experiment and finally a conclusion is drawn out for each experiment.

4.2 Experimental Environment Setup

Since our main focus is on the Resource Optimization Property Manager (ROPM) as a proof of concept, our experiments were all focused on this model of the implemented prototype. All experiments were conducted on Microsoft Windows 2000 professional and Windows XP professional as the testing platform. We also used the IBM Apache HTTP Server 1.3.26 [5] for windows as the HTTP experiment based server and IBM WebSphere Application Server 5.0 [38] for windows as the experiment base application server. Both servers are available as free trial versions on IBM website [37]. In our prototype implementation we used the Autonomic Manager Tool Set (AMTS) which is part of the Emerging Technologies Toolkit (ETTK 1.1) provided by IBM on their website [15] also as a free download trial version. The Autonomic Manager toolset (AMTS) is designed to aid in the creation of the management portion of Autonomic Elements. The toolset can be used to build a wide variety of elements, ranging from simple components that monitor low level resources, to complete autonomic elements managing large complexes of distributed computing middleware. In this toolset a set of components are provided which can be used in a variety of ways to yield many different solutions. The detailed documentation of the toolset can be found on [15]. The java programming language was used in the prototype development in order to make use of its portability feature in addition to the other known benefits of the java programming language as an object oriented programming language. We used JSDK 1.3.1 [22] for the JVM implementation and the sun Java Messaging Server (JMS) (included in j2sdkee1.3.1 [21]) to be the communication server between the different property manager clients and the property manager server itself. As for the programming IDE, we used Oracle JDeveloper 9i [23], which is freely provided by Oracle. Finally, we

used the Open System Testing Architecture (OpenSTA 1.4) [32], which is an open source tool, as the Application Server/ HTTP server stress testing and load generation tool. The details of using this tool are found on [32]. We conducted different experiments to simulate the different scenarios for the major features of the resource optimization property manger (ROPM) in addition to one experiment, which included a simulated interaction between the security and optimization property manager. For simplicity, from this point onwards we will refer to the Resource Optimization Resource Manager as ROPM.

All of the performed experiments were conducted using three different machines. All of the machines platforms used the Internet Explorer 6.0 as a web browser. The first machine runs windows XP professional. It has an Intel based PIII 750 processor with 512 MB of RAM. This machine is referred to as the “Client Machine” throughout the different experiments. The second machine, which we will always refer to as the “Server Side Machine”, has a P III 900 Intel based processor and 640 MB of RAM and runs windows 2000 professional based OS. The third machine, which is referred to as “The probing station machine”, runs windows 2000 professional OS based on PII 333 Intel processor with 128 MB of RAM. Each machine has a 10/100 Mbps Fast Ethernet card and they were all connected using a 10/100 Mbps switch.

4.3 Experiment One

4.3.1 Experiment Objective

The objective of this experiment is to simulate two basic scenarios. The first scenario simulates the normal behavior of the IBM apache HTTP server’s performance and improvement in response to changing the value of the apache tuning parameter “ThreadsPerChild” which controls the number of concurrent worker processes that handle the incoming HTTP requests. The second scenario simulates the high level CPU utilization effect on the apache server average response time as the tuning parameter “ThreadsPerChild” is incremented. The value associated with the tuning parameter “ThreadsPerChild” in the “httpd.conf” file, (one of the apache configuration files) indicates the number of concurrent worker processes that the apache server will create to serve the incoming requests. In theory as the number of “ThreadsPerChild” is increased, the response time of the apache server

improves since the average waiting time of the incoming requests will decrease and consequently the end user average response time will improve. However in this experiment we prove that there are other controlling factors in addition to the number of worker processes that can affect the response time. Because of the close relationship between this tuning parameter and the performance metrics of the machine on which the apache server runs (CPU and Memory usage in our case), changing this parameter will proportionally affect these metrics (i.e. increasing the number of the apache concurrent threads usually means an increase in the CPU and Memory consumption). In fact this change imposes a limitation on the response time improvement that can be reached with respect to the increase in the number of concurrent threads that are bound to the CPU and Memory utilization levels as will be illustrated in Test 2 of this experiment. As the available apache server version does not support dynamic parameter change recognition (i.e. the change in the tuning parameter “ThreadPerChild” during run time, is not done by the Apache server until it is restarted), we had to restart the Apache server during some of the mentioned tests each time we changed the tuning parameter. Restarting the apache server is a different process from stopping and starting the server over again. Restart is a less harmful process since it actually pauses the server and refreshes most of the configuration parameters and resumes the server operation. Actually this work around slightly affected the test results by appearing as sharp edge drops in the server response time in most of the resulting graphs. However this did not affect the total results of the performed tests. Most of the conducted tests in this experiment were run for an average period of two to three minutes.

4.3.2 Test Case Environment Setup & Result Analysis

This experiment consists of three tests. In all the tests, we used the stress and load generating tool (OpenSTA) to generate a load by a number of virtual users which run a set of recorded scripts. Each recorded script simulates a set of requests for static pages, which reside in the apache server. We used the default manual pages provided by the apache server as the script based static pages.

4.3.3 Test One

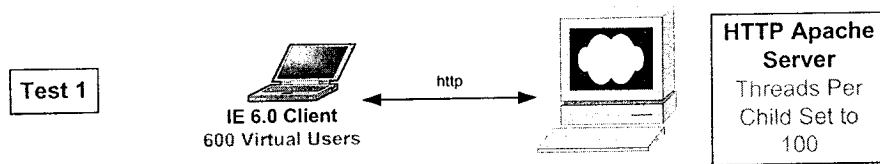


Figure 4.1: Apache Server Test 1

In this test the apache server tuning parameter “ThreadsPerChild” was adjusted to 100 concurrent threads and a stress load of 600 simulated concurrent virtual users were generated from the client side. The test ran for a period of almost two minutes. The overall test setup is illustrated in figure 4.1.

The graph in figure 4.2 reflects the number of virtual users used during the test elapsed time (i.e. the graph reflects the load imposed on the apache server). The maximum number of users reached during the test as seen in the Y axis is 600 concurrent virtual users. The graph in figure 4.3 reflects the HTTP requests response time in milliseconds against the test elapsed time. We observe that the response time reaches a peak of about 9500 ms at the beginning of the test and then fluctuates around 2000 ms during the rest of the test and reaches a zero at the end of the test and that is where the requests are completely stopped. The explanation for the high response time at the beginning of the test is that the rate at which the 600 virtual users are started is relatively high and this causes a sort of a panic state for the apache server during the very few seconds of the test.

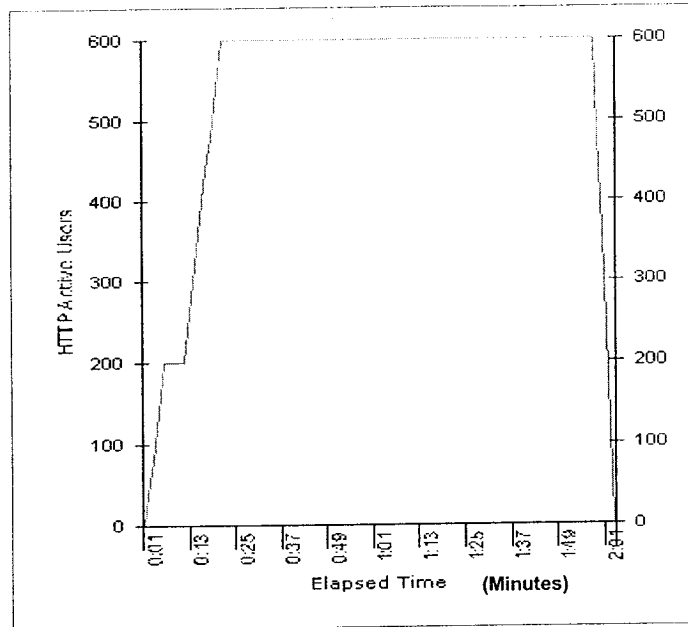


Figure 4.2: Active Users Vs Test Elapsed Time

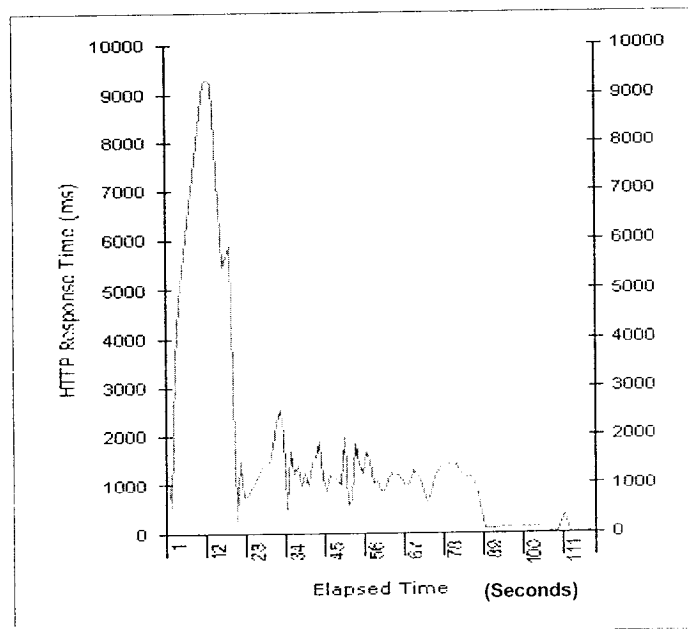


Figure 4.3: Response Time Vs Test Elapsed Time

The graph in figure 4.4 indicates the number of HTTP returned errors against the number of concurrent HTTP requests. We notice that the number of errors reaches a peak of about 400 errors per 380 concurrent requests. The graph in figure 4.5 reflects the number of HTTP errors during the test elapsed time and we notice the sharp upslope curve, which started after almost 100 seconds of the test starting time.

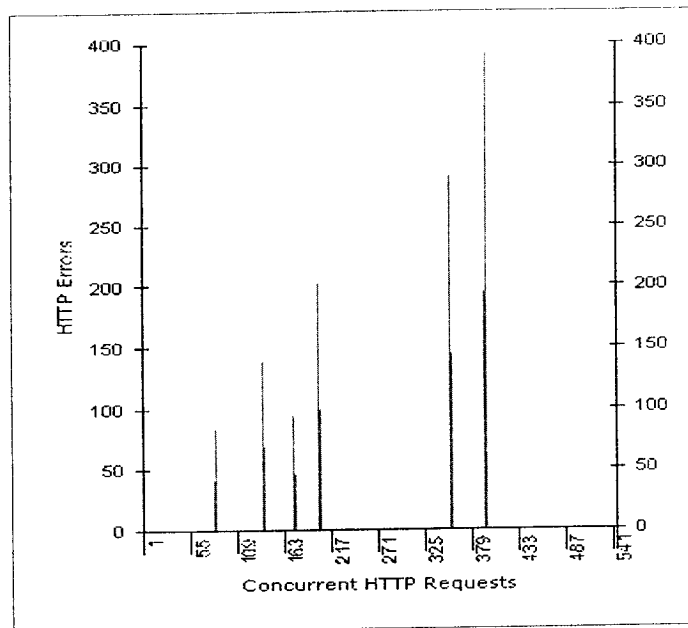


Figure 4.4: HTTP Errors Vs Concurrent HTTP

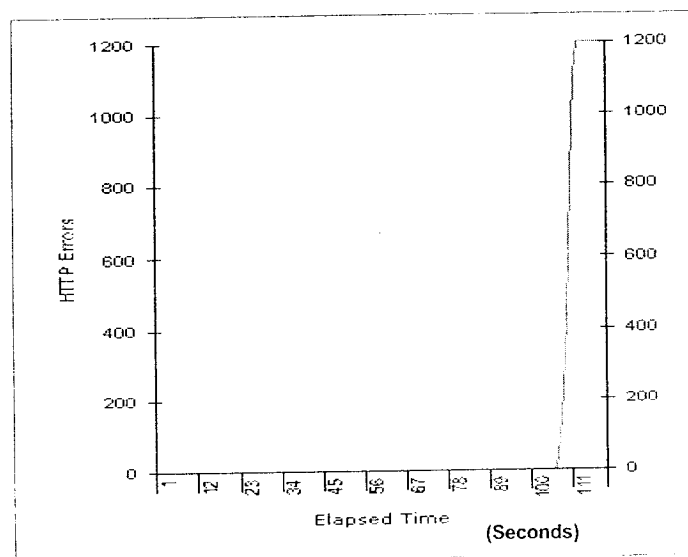


Figure 4.5: HTTP Errors Vs Elapsed Time

4.3.4 Test Two

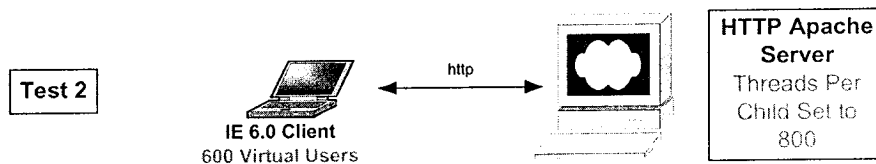


Figure 4.6: Test Environment Setup

Test two runs with the same settings as test one, but we increased the number of the apache tuning parameter “ThreadsPerChild” up to 800 concurrent worker process in order to improve the response time.

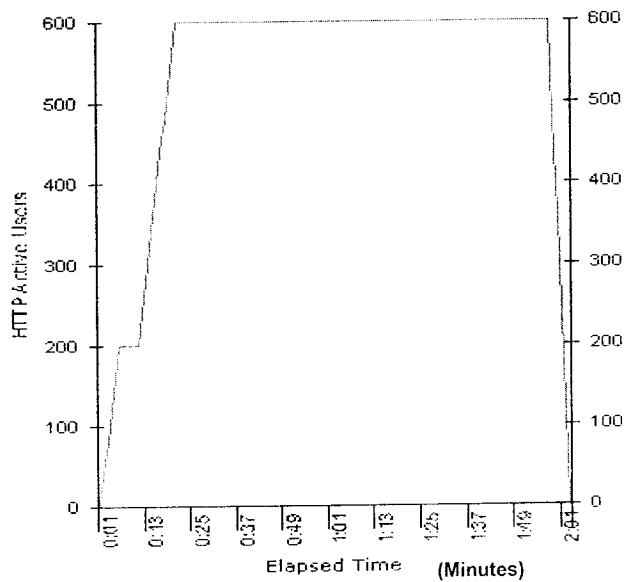


Figure 4.7: HTTP Active Users Vs Elapsed Time

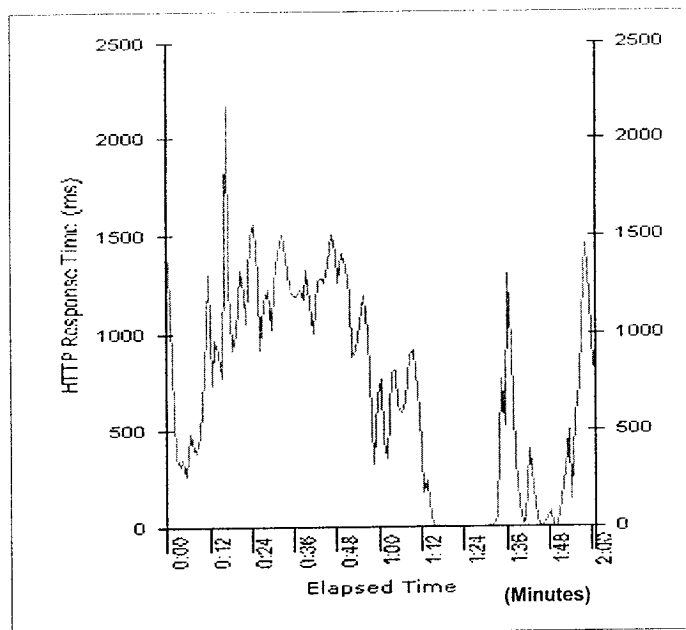


Figure 4.8: HTTP Response Time Vs Elapsed Time

The graph in figure 4.7 reflects the number of virtual users during the test elapsed time (load graph), whereas the graph in figure 4.8 reflects the HTTP response time in ms during the test elapsed time. We notice here the improvement in the response time shown in this graph relative to the one shown in figure 4.3 of test one. The maximum response time shown in figure 4.8 reaches a value of nearly 2000 ms and the rest of the test fluctuates within the range of 1000 to 1500 ms, whereas the peak value of the response time in test one had reached 9000 ms and the average value was about 2000 ms.

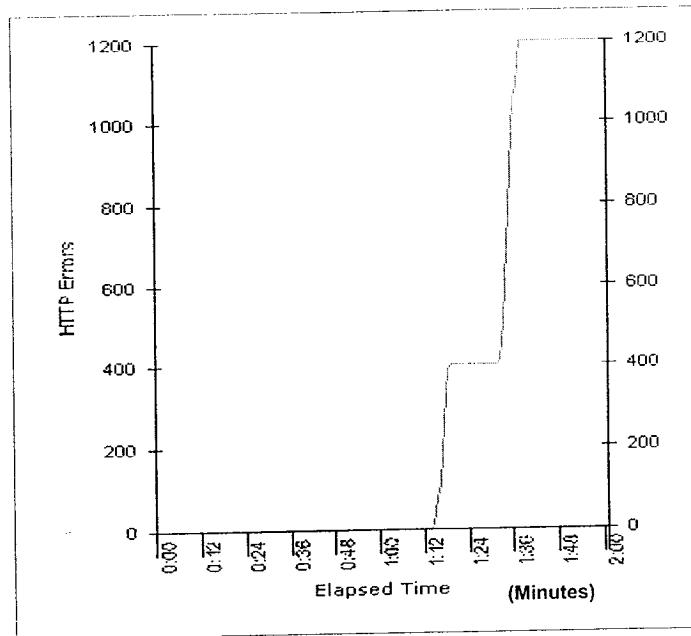


Figure 4.10: HTTP Errors Vs Elapsed Time

In conclusion, as we increase the number of “ThreadsPerChild” in the Apache HTTP Server, we obtain a better response time and a decreased number of errors for a higher number of concurrent requests. Thus keeping the CPU load on the hosting machine at a constantly low load rate.

4.3.5 Test Three

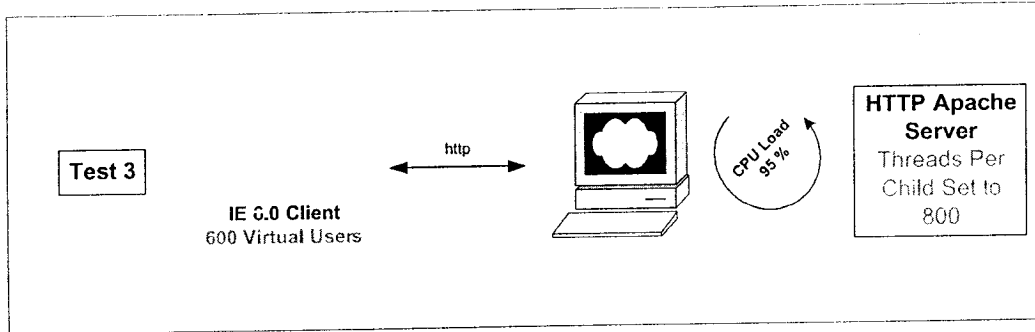


Figure 4.11: Test Three Environment Setup

In this test, as shown in figure 4.11, we used the same settings and test case scenario as the one used in Test 2. Additionally we increased the CPU imposed load during the experiment elapsed time to reach a CPU utilization level of about 95%. We noticed that both the average response time, and the number of errors against the number of concurrent HTTP requests has increased. We used a program that runs a number of infinite loops to consume most of the CPU time during the test elapsed time and in this way we simulated a synthetic high CPU utilization. We also raised the tuning parameter “ThreadsPerChild” from (100) to (800) after the first minute of the test, thus keeping a constant load of (600) virtual users during the whole test. The whole test ran for an average period of 2 minutes.

Figures 4.12 and 4.13 present two important graphs which provide us with some useful information concerning the performance behavior of the Apache server in response to the increment in the number of working threads (i.e. changing the tuning parameter “ThreadsPerChild”) during an excessive CPU utilization on the running machine. The graph in figure 4.12 reflects the server response time during the different phases of the test. As we notice after the first minute of the test running time, the apache server was restarted to increase the number of worker threads “ThreadsPerChild”. Actually this is reflected in the response time curve’s sharp edge drop down at the value of nearly 1 minute on the x-axis. What causes the sharp and sudden incline in the response time curve is that the server goes into the panic state for a few seconds after it had just started up. However a few seconds later, it stabilizes near the value of 10000 milliseconds. So despite the fact that we increased

the tuning parameter value of “ThreadsPerChild”, we got poorer response time in comparison to the one we got in test 2.

The graph in figure 4.13 reflects the number of generated HTTP errors against the number of Concurrent HTTP requests. It also indicates the increase in the error numbers that are generated during the test elapsed time relative to the number of errors generated in test two and shown in figure 4.9. In the case of test two the number of errors reached a peak of 180 errors per 305 concurrent requests, whereas in this test it reached a maximum value of 250 errors per 260 concurrent requests. Averaging the increment in the number of errors, we reach an average percentage increment of about 39% increase. Hence, what the two graphs indicated is the effect that the high CPU utilization have on the apache server performance improvement despite the positive change in the “ThreadsPerChild” tuning parameter.

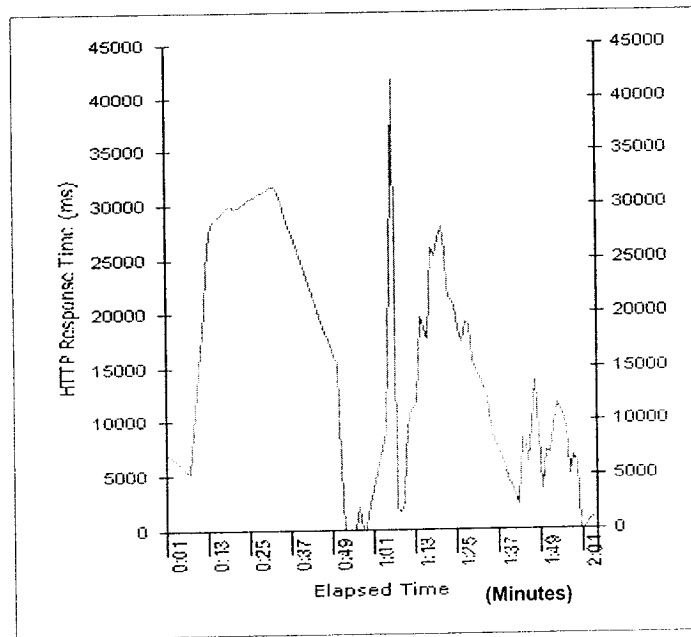


Figure 4.12: HTTP Response Time Vs Elapsed Time

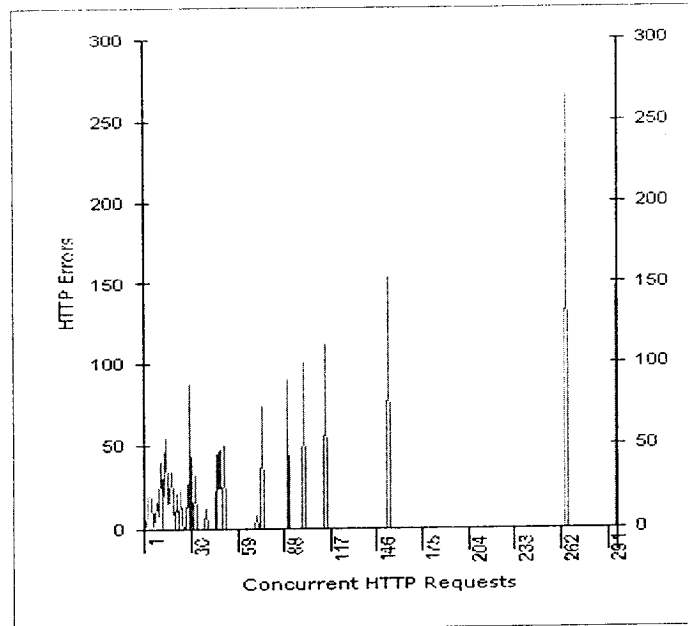


Figure 4.13: HTTP Errors Vs Concurrent HTTP Requests

4.3.6 Experiment Conclusion

Tests 1 and 2 of this experiment have shown that the normal behavior of the HTTP Apache server as simulated in our testing environment, is important to verify that all the subsequent experiments' results would also simulate a real environment outcome. Test 2 mainly demonstrated that as we increase the number of "ThreadsPerChild" in the Apache HTTP Server, we obtain a better response time and a reduced number of errors for a higher number of concurrent requests provided that the CPU utilization load on the hosting machine is not too high. The purpose of test three was to highlight the effect of the CPU utilization on the Apache server during the change in the tuning parameter "ThreadsPerChild". We have shown that increasing the number of serving threads of the apache server during a high CPU utilization will not have the same effect as when it is done during a normal CPU utilization load. This was obvious in the average response time and number of HTTP errors of test 3 in comparison to test 1 & 2, since the gained improvement in the server response time was not immense due to the high level of CPU utilization.

The Auto-Tune agents project which was conducted by a group of researchers [41] from the IBM research labs, was concerned with studying the problem of controlling CPU

and memory utilization of an Apache ® Web server [5] using the application-level tuning parameters MaxClients and KeepAlive, which were exposed by the server. However, their dynamic model is based on the assumption that the system administrator will provide fixed values for the CPU and memory utilization levels. The MaxClients parameter is the exact equivalent of the “ThreadsPerChild” tuning parameter in our experiments. In fact, there is more than one concern about this assumption. One of these concerns is about the way in which we can guarantee that the server will be able to reach the level of specified utilization, since it can only control its process utilization and does not have control over the rest of the systems sharing the resources. Another concern is about changing the required utilization levels dynamically according to the current system need. In other words, the apache utilization model might have been set to a certain utilization level that requires a change in the tuning parameter value “ThreadsPerChild” whereas the current available resources do not allow the achievement of this value. Consequently the apache utilization level should be readjusted in accordance to the available resources. In the given model [41], the administrator has to specify the required utilization levels statically and then the dynamic model embedded in their system changes the tuning parameters to reach the specified utilization levels. Hence we could see that model lacks the flexibility of changing the utilization levels dynamically and according to the need. In the coming experiments will illustrate how the Autonomic Property Manager will help in overcoming some of these problems.

4.4 Experiment Two

4.4.1 Experiment Objective

In this experiment we present a demonstration for the use of the Resource Optimization Property Manager prototype. The experiment demonstrates that according to a predefined set of policies that are pre-set by the system administrator inside the policy definition files, the system responds to the specified sensors evaluation and triggers the proper policy actions/effectors. As was mentioned earlier in the design section, policies are normally defined at different levels in order to abstract the business layer. For example in this experiment there are two levels for the policy definition. At the first high level, we define a WebSite policy which includes the definition of the desired response time threshold specified by the business need regardless of all the subsequent layers. The policy files for this level includes the measurement of the average response time from a probing station against a defined threshold value which fires a certain policy action whenever this level is exceeded. The action section for this policy includes a call to the next policy definition level, which in our case is the HTTP server policy. Figure 4.14, and Figure 4.15 present a snapshot for the WebSite policy precondition and action sections.

The HTTP policy defines two event driven sensors, CPU usage reading sensor and an apache tuning parameter reader sensor (that reads the “ThreadsPerChild” value in our case). A CPU usage threshold value is also defined in the system policy. The corresponding sensor reading should always be below this value otherwise the related action policy will be fired. In addition to a threshold value for the retrieved tuning parameter a value for the Apache server (ThreadsPerChild) is also defined. The action part of the policy includes a call to a remote system function which increases the apache tuning parameter value “ThreadsPerChild” as required and then restarts the server in order for it to be recognized by the server. Figure 4.16 and figure 4.17 present a snapshot for the HTTP Policy definition and action sections, respectively.

```

<?xml version="1.0" encoding="UTF-8" ?>
- <PerformanceMonitorPolicyDocument version="1.0" discipline="PerformanceMonitorPolicy">
  <!-- ... -->
  <!-- ... -->
  - <ConditionSection>
    - <SimpleCondition policyElementId="c1">
      <Operator>greaterThan</Operator>
      - <Operand1>
        - <Expression>
          <VariableName>measuredValue</VariableName>
        </Expression>
      </Operand1>
      - <Operand2>
        - <Expression>
          <VariableName>expectedResponseTime</VariableName>
        </Expression>
      </Operand2>
    </SimpleCondition>
  </ConditionSection>
+ <ActionSection>
+ <ACPolicySection>
- <ACPolicyGroup policyElementId="g1">
  <ACPolicyRef name="p1" />
</ACPolicyGroup>
</PerformanceMonitorPolicyDocument>

```

Figure 4.14: Web Policy Precondition Section

```

+ <ConditionSection>
- <ActionSection>
  - <SimplePolicyAction policyElementId="a1">
    - <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
      - <Parameter>
        - <Expression>
          <Literal>"Policy"</Literal>
        </Expression>
      - <Expression>
        <Literal>"HTTP"</Literal>
      </Expression>
    </Parameter>
  </MethodExpression>
</SimplePolicyAction>
</ActionSection>
- <ACPolicySection>
  - <ACPolicy policyElementId="p1">
    - <Precondition>
      <SimpleConditionRef>c1</SimpleConditionRef>
    </Precondition>
    <SimplePolicyActionRef>a1</SimplePolicyActionRef>
  </ACPolicy>
</ACPolicySection>
<ACPolicyGroup policyElementId="g1">
  <ACPolicyRef name="p1" />
</ACPolicyGroup>
</PerformanceMonitorPolicyDocument>

```

Figure 4.15: Web Policy Action Section

```

- <OFExpression policyElementId="HTTPServerNumOfProcessProc_CPU">
- <SimpleCondition policyElementId="c1">
  <Operator>lessThan</Operator>
  - <Operand1>
    - <Expression>
      <VariableName>NumberOfProcess</VariableName>
    </Expression>
  </Operand1>
  - <Operand2>
    - <Expression>
      <Literal>100</Literal>
    </Expression>
  </Operand2>
</SimpleCondition>
</OFExpression>
- <OFExpression policyElementId="HTTPServerCPU">
- <SimpleCondition policyElementId="c1">
  <Operator>lessThan</Operator>
  - <Operand1>
    - <Expression>
      <VariableName>CPU_USAGE</VariableName>
    </Expression>
  </Operand1>
  - <Operand2>
    - <Expression>
      <Literal>30</Literal>
    </Expression>
  </Operand2>

```

Figure 4.16: HTTP Policy Precondition Section

```

- <ActionSection>
- <SimplePolicyAction policyElementId="a1">
  - <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
    - <Parameter>
      - <Expression>
        <Literal>"FunctionCall"</Literal>
      </Expression>
    </Parameter>
    - <Expression>
      <Literal>"void|PropertyManager.ApacheRestart|ReplaceText|ThreadsPerChild,400,
Files\\IBMHttpServer\\ApacheManualRestart.bat,|;"</Literal>
    </Expression>
  </MethodExpression>
</SimplePolicyAction>
</ActionSection>
- <ACPolicySection>
- <ACPolicy policyElementId="p1">
  - <Precondition>
    <OFExpressionRef>ServerProcessNum_CPUusage_Cond</OFExpressionRef>
  </Precondition>
  <SimplePolicyActionRef>a1</SimplePolicyActionRef>
</ACPolicy>
</ACPolicySection>
- <ACPolicyGroup policyElementId="g1">
</PerformanceMonitorPolicyDocument>

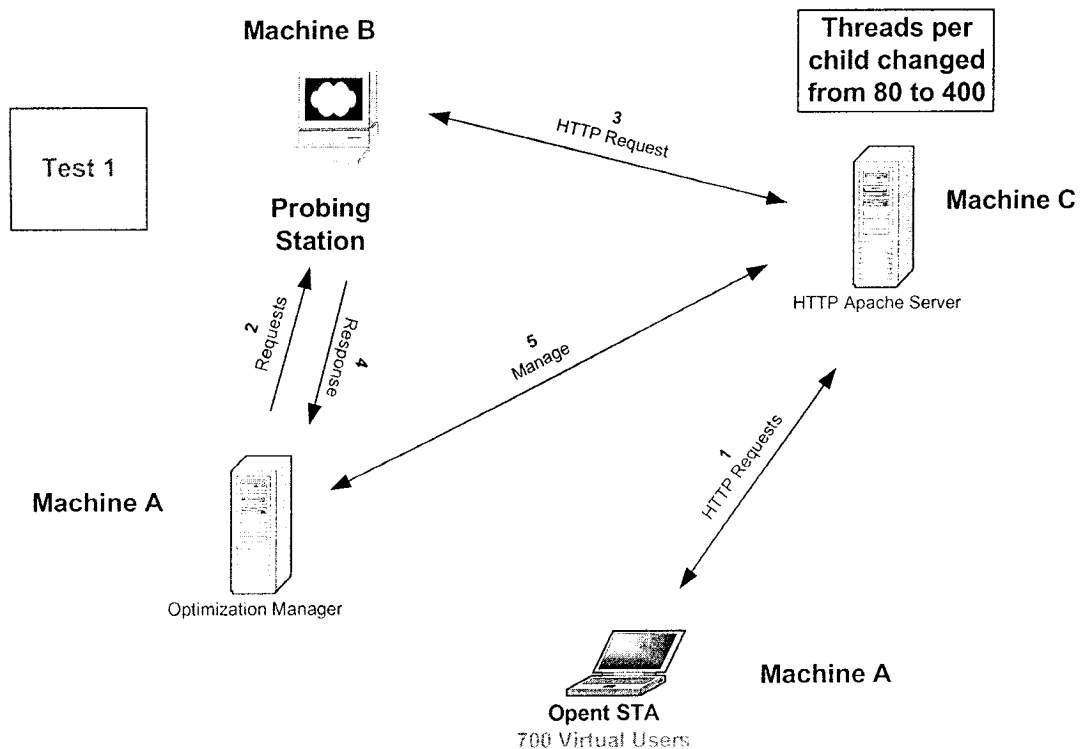
```

Figure 4.17: HTTP Policy Action Section

4.4.2 Test Case Environment Setup & Result Analysis

As shown in figure 4.18, we used three different machines in this test (Named machine A, B, and C). Machine A hosts the Resource optimization manager and the stress load generation tool (OpenSTA). Machine B hosts the probing station. Machine C hosts the Apache HTTP server and a resource optimization manager client to receive the incoming messages. In this test the stress testing tool starts to generate a load of 700 concurrent virtual users as shown in step one. As the resource optimization manager is constantly calling the probing station to evaluate the returned response time value against the defined threshold (steps 2,3,4), it finds out that the returned value exceeds the threshold and consequently it fires the defined policy action which increments the apache server tuning parameter “ThreadsPerChild” (step 5). After a while the response time returns a value below the defined threshold, and consequently the policy action is never triggered again.

Figure 4.18: Test 1 Environment Setup



The graph in figure 4.19 indicates the number of virtual users during the test elapsed time (load graph) and we notice that it has reached a peak value of 700 concurrent users. The graph in figure 4.20 reflects the response time improvement as a result of incrementing the apache tuning parameter “ThreadsPerChild” from 80 to 400. In fact we can notice that this action has taken place almost three minutes after the test start time. While the whole test period remained for about six minutes. What actually happened is that the Resource Optimization manager detected a value for the probing station sensor (that actually represents the apache HTTP server response time), which exceeds the defined threshold value. As a

result, it fired the corresponding policy action which in our case has incremented the apache tuning parameter “ThreadsPerChild” and resulted in a better response time value.

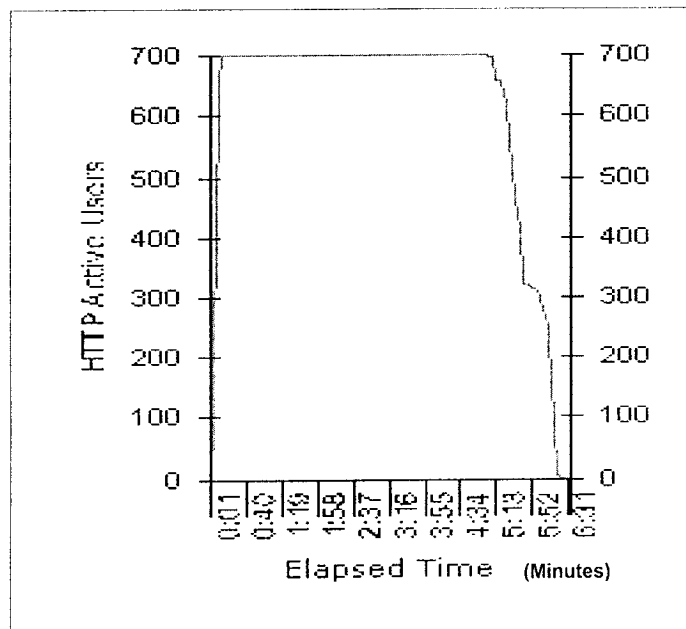


Figure 4.19: HTTP Active Users Vs Elapsed Time

The graph in figure 4.19 indicates the number of virtual users during the test elapsed time (load graph) and we notice that it has reached a peak value of 700 concurrent users. The graph in figure 4.20 reflects the response time improvement as a result of incrementing the apache tuning parameter “ThreadsPerChild” from 80 to 400. In fact we can notice that this action has taken place almost three minutes after the test start time. While the whole test period remained for about six minutes. What actually happened is that the Resource Optimization manager detected a value for the probing station sensor (that actually represents the apache HTTP server response time), which exceeds the defined threshold value. As a result, it fired the corresponding policy action which in our case has incremented the apache tuning parameter “ThreadsPerChild” and resulted in a better response time value.

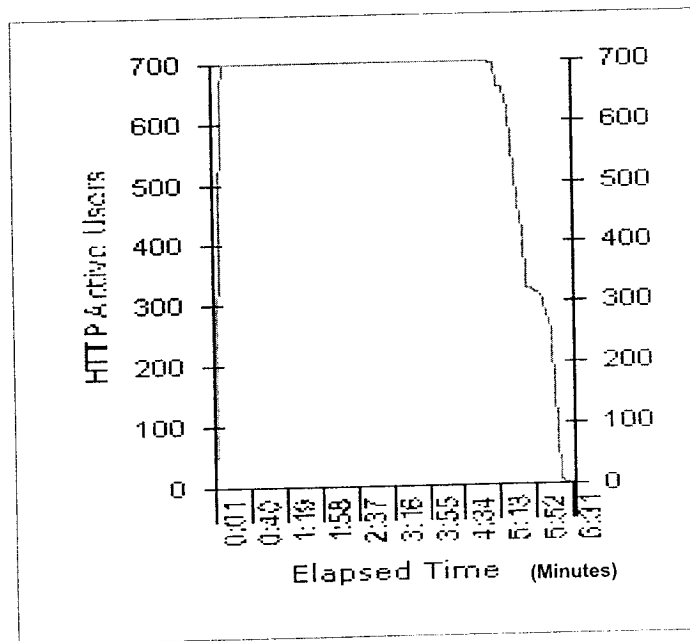


Figure 4.19: HTTP Active Users Vs Elapsed Time

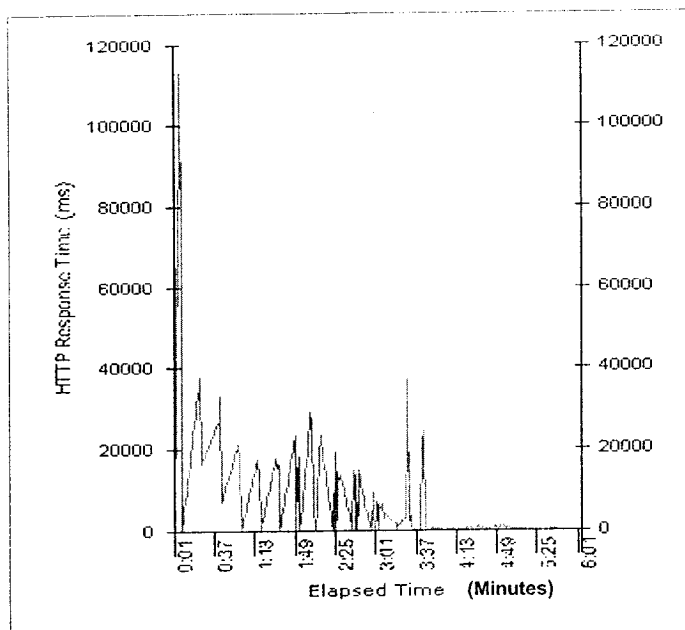


Figure 4.20: HTTP Response Time Vs Elapsed Time

4.4.3 Experiment Conclusion

As mentioned earlier, this experiment demonstrated the capability of the Resource Optimization Property Manager (ROPM) to make decisions and corrective actions based on a predefined set of policies, which include threshold values definition for a set of different parameters/sensors. This experiment also illustrated the capability of the property manager to make calls for external tools, built functions, external functions as we did for the CPU usage utilization measurement sensor (built in function), the probing station sensor (external function call), and finally executing the effector which actually called on a tool to execute the apache server parameter adjustment (external function call in addition to an apache tool call). We have also demonstrated the capability of a hierarchical policy call and high level policy definition represented in the WebSite and HTTP policy definitions. The very high level goal of the system (Website response time) was defined inside the WebSite policy, and that policy was able to call other lower level policies when the set precondition (exceeding the threshold value) was violated. We recall here the Auto-Tune agents project [1] that has a limitation on the scope of system monitoring which is bound to the apache server process and restricted resource control or reallocation. In our model the CPU utilization level can be changed

dynamically by invoking the effectors within the action policy section in addition to the higher scope of resource monitoring and utilization that can be obtained by the ROPM for all the managed systems. Thus a better level of resources manipulation and utilization can be achieved and that will be further illustrated in the coming experiments.

4.5 Experiment Three

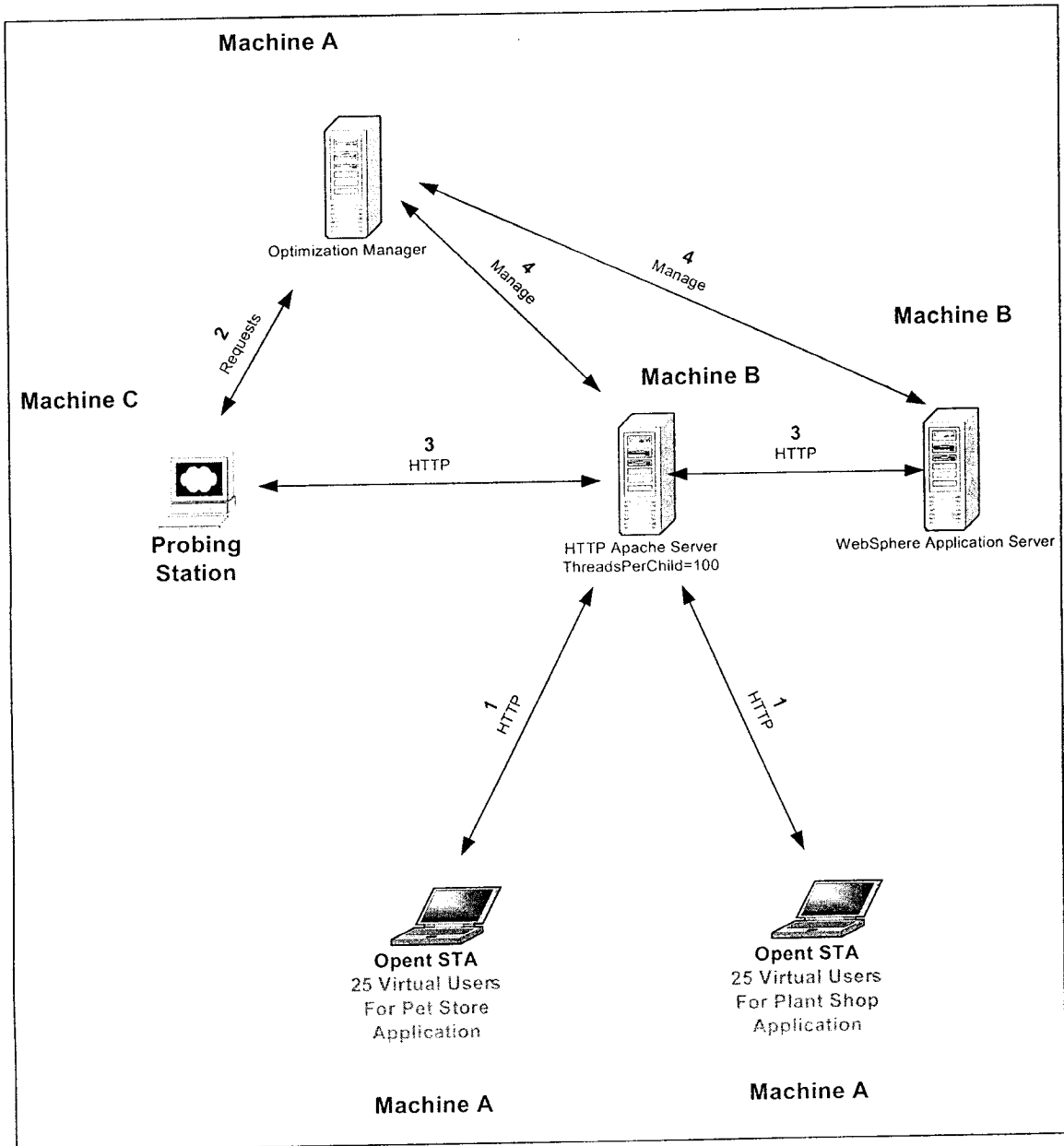


Figure 4.21: Experiment Three Environment Setup

4.5.1 Experiment Objective

The purpose of this experiment is to demonstrate the way in which we can make use of some of the provided features by the resource optimization property manager. In this experiment the resource allocation manager was triggered by one of the policies when extra resource allocation was needed. The job of the resource allocation manager is to check the feasibility of allocating a percentage of the needed resources at the request time by revisiting all the registered systems current consumption. The detailed mechanism of the resource allocation manager is mentioned in the design section. At the beginning of this experiment the HTTP policy is triggered as a result of the returned feedback value of the probing station sensor. The HTTP policy concludes the need for extra CPU resource allocation. Consequently the resource allocation server is consulted and by going through the full logic cycle of the resource allocation, it decides to temporarily stop one of the WebSphere running web applications (Pet Store application) which was consuming a great amount of the CPU time, while it has a lower priority and was operating in its off peak time. So this application is stopped, and more CPU resources were released. As a result, an improvement in the response time and number of errors of the monitored node (Plant Shop) is reflected.

4.5.2 Test Case Environment Setup & Result Analysis.

As shown in figure 4.21, we used three machines (Named machine A, B, and C). Machine A hosts both the Resource Optimization Property Manager and the stress generation tool. Machine B hosts the Apache HTTP server and the WebSphere Application server. Machine C hosts only the probing station. We used the two sample web applications Pet Store and Plant Store, which are freely shipped with the IBM WebSphere application server. Both applications resemble E-Commerce shopping stores. As their names indicate one application serves as an E-Pet shopping store and the other serves as a E-Plant shopping store. Detailed descriptions for the sample application structures could be found in the WebSphere application server documentation. In this test we used a set of generated scripts, which simulate the user actions during the E-shopping process using each of the E-stores. Each application runs on a separate web application container that can be started, stopped, restarted, and it consumes some of the resources allocated for the Application server. Consequently it affects the performance of the rest of the applications as they are served by

one Application server and share the same resources. We can notice in figure 4.21 that a load of 25 virtual users is generated for each of the Pet Store and the Plant store concurrently. In order to accurately calculate the response time of each application, we have separately measured the metrics of each application.

As the Resource Optimization Property Manager calls the probing station sensor to measure the response time of the Plant store application, it finds out that the response time exceeds the defined threshold in the Web Policy file, so it fires the HTTP policy as the defined effector. As the ROPM evaluates the preconditions of the HTTP policy, it matches one of the defined preconditions (a snapshot of the preconditions is shown in figure 4.22), which indicates that the CPU usage utilization level has exceeded 75% and that the number of the apache concurrent threads “ThreadsPerChild”, is set to a value higher than 80 processes. Accordingly, it fires the corresponding actions, which are shown in figure 4.23 (the fired action is defined as action “a2” in figure 4.23). This action consults the resource allocation manager to request an allocation for additional CPU resources. The resource allocation manager in this experiment decides to shutdown one of the Web Applications hosted by the application server which is the Pet store in our case as it is found to be the least priority running application with the highest CPU resource consumption. As the pet store application stops, an improvement in the Plant store response time is sensed and consequently the required response time is achieved.


```

- <OFExpression policyElementId="HTTPServerNumOfProcessProc_CPU">
- <SimpleCondition policyElementId="c2">
  <Operator>greaterThan</Operator>
  - <Operand1>
    - <Expression>
      <VariableName>NumberOfProcess</VariableName>
    </Expression>
  </Operand1>
  - <Operand2>
    - <Expression>
      <Literal>80</Literal>
    </Expression>
  </Operand2>
</SimpleCondition>
</OFExpression>
- <OFExpression policyElementId="ServerCPU">
- <SimpleCondition policyElementId="c2">
  <Operator>greaterThan</Operator>
  - <Operand1>
    - <Expression>
      <VariableName>CPU_USAGE</VariableName>
    </Expression>
  </Operand1>
  - <Operand2>
    - <Expression>
      <Literal>75</Literal>
    </Expression>
  </Operand2>
</SimpleCondition>
</OFExpression>

```

Figure 4.22: Policy Precondition Section

```

- <SimplePolicyAction policyElementId="a1">
- <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
  - <Parameter>
    - <Expression>
      <Literal>"FunctionCall"</Literal>
    </Expression>
  </Parameter>
  - <Expression>
    <Literal>"void | PropertyManagerPack.ApacheRestart | ReplaceText | ThreadsPerChild,400
    Files\\Apache Group\\Apache2\\ApacheManualRestart.bat, |;"</Literal>
  </Expression>
</MethodExpression>
</SimplePolicyAction>
- <SimplePolicyAction policyElementId="a2">
- <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
  - <Parameter>
    - <Expression>
      <Literal>"FunctionCall"</Literal>
    </Expression>
  </Parameter>
  - <Expression>
    <Literal>"BuiltIn | ResourceAllocator | AllocateResource | CPU,HTTPServer,true, |;"</Literal>
  </Expression>
</MethodExpression>
</SimplePolicyAction>
</ActionSection>

```

Figure 4.23: Policy Action Section

The graph in figure 4.24 depicts the number of virtual users during the test elapsed time, whereas the graph in figure 4.25 presents the Plant shop response time improvement right after the Pet store application was stopped which is almost one minute after the stress test had started. The previously mentioned conclusion can be observed in figure 4.25, where a drop in the response time curve took place after almost one minute to change from an average response time of 6000 ms to an average response time of 3000 ms. Figure 4.26 depicts the response time depreciation that took place in the Pet store response time and that ends up at a zero response time after almost 1 minute as a result of the application complete stop. This fact is also reflected in Figure 4.27 where it illustrates the incremental number of HTTP errors returned from the Pet store during the test elapsed time which has reached a peak of 2000 and again that indicates the complete stop of the application server.

4.5.3 Experiment Conclusion

This experiment illustrates the capability of the ROPM to provide some built in functions such as resource allocation consultancy, which can be used inside the policy definitions when needed. Actually this experiment demonstrates the dynamic capabilities that can be provided by the ROPM in detecting one of the run time optimization problems and enables it to solve it dynamically by using the predefined set of policies. In fact this model is not restricted to the optimization property manager only. Any other property manager can actually use it, and this is the whole point of the autonomic computing work. It is the system capability of self-management (i.e. it detects the violations and it initiates the appropriate actions to handle those violations) and this is what this experiment proves.

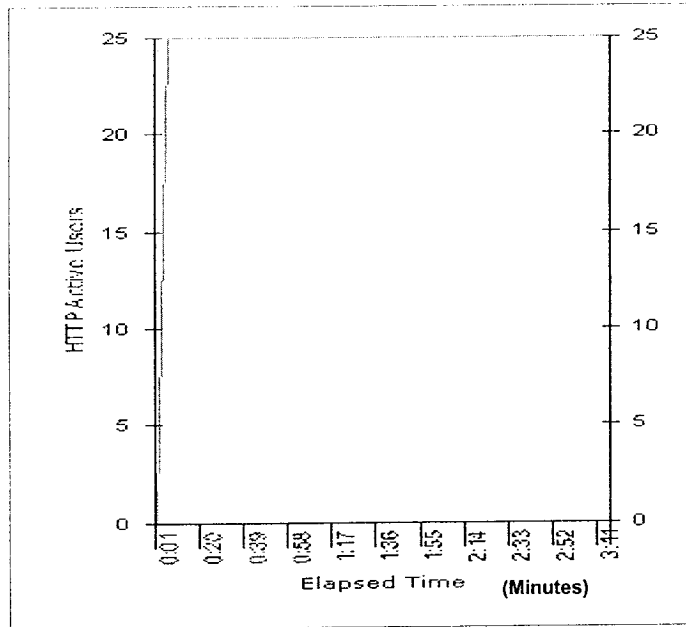


Figure 4.24: HTTP Active Users Vs Elapsed Time

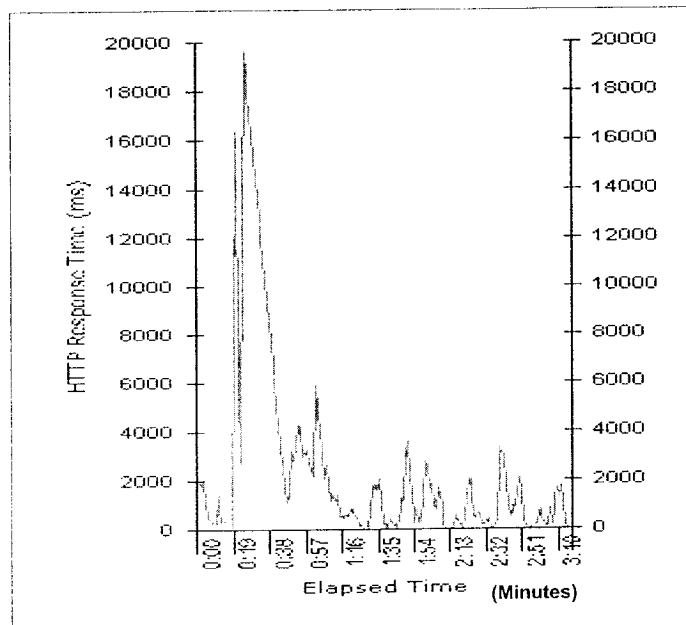


Figure 4.25: HTTP Response Time Vs Elapsed Time

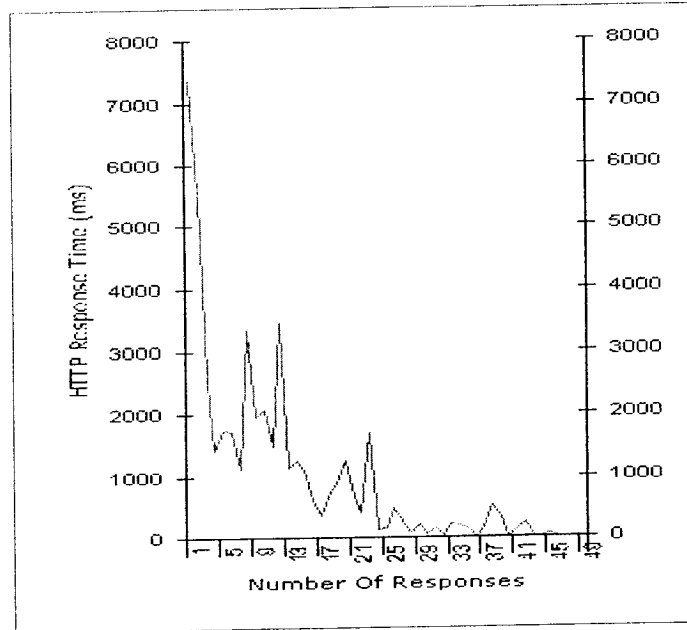


Figure 4.26: Response Time Vs Number Of Responses



Figure 4.27: HTTP Errors Vs Elapsed Time

For simplicity what the security manager actually does in this experiment is that it sleeps for sometime to simulate the task of performing some actions, which will consume sometime. After a while, it releases the global lock and sets its threshold value for the probing station sensor to a very high value so that the policy precondition is never satisfied again. Figure 4.28 presents a snapshot of the security policy precondition, which evaluates the probing station's measured value sensor against the defined security threshold value. Figure 4.29 presents a snapshot of the action policy section where the security manager unlocks the global lock by setting its value to zero and then sets the security threshold value to an infinite value so that the action is never triggered again.

```

<?xml version="1.0" encoding="UTF-8" ?>
- <SecurityPolicyDocument version="1.0" discipline="SecurityPolicy">
  <!-- ... .. -->
  <!-- ... .. -->
  - <ConditionSection>
    - <SimpleCondition policyElementId="SecurityCond">
      <Operator>greaterThan</Operator>
      <Operand1>
        - <Expression>
          <VariableName>measuredValue</VariableName>
        </Expression>
      </Operand1>
      <Operand2>
        - <Expression>
          <VariableName>SecurityThreshold</VariableName>
        </Expression>
      </Operand2>
    </SimpleCondition>
  </ConditionSection>
+ <ActionSection>
  - <ACPolicySection>
    + <ACPolicy policyElementId="SecurityPol">
    </ACPolicySection>
  - <ACPolicyGroup policyElementId="SecurityGroup">
    <ACPolicyRef name="SecurityPol" />
  </ACPolicyGroup>
</SecurityPolicyDocument>

```

Figure 4.28: Security Policy Precondition Section

```

- <ActionSection>
  - <SimplePolicyAction policyElementId="SecurityAct">
    - <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
      - <Parameter>
        - <Expression>
          <Literal>"FunctionCall"</Literal>
        </Expression>
        - <Expression>
          <Literal>"BuiltIn|PolicyServer|SetKBVariable|GlobalLock,0,|;"</Literal>
        </Expression>
      </Parameter>
    </MethodExpression>
  </SimplePolicyAction>
  - <SimplePolicyAction policyElementId="SecurityThresholdAct">
    - <MethodExpression MethodName="sendAlertEffector" ReturnType="void">
      <Parameter>
        - <Expression>
          <Literal>"FunctionCall"</Literal>
        </Expression>
        - <Expression>
          <Literal>"BuiltIn|PolicyServer|SetKBVariable|SecurityThreshold,9999,|;"</Literal>
        </Expression>
      </Parameter>
    </MethodExpression>
  </SimplePolicyAction>
</ActionSection>

```

Figure 4.29: Security Policy Action Section

4.6.2 Test Case Environment Setup & Result Analysis

Figure 4.30 illustrates the experimental run sequence and test environment settings. The environment setup is the same as experiment three's setup. Additionally an instance of the security property manager is running on the same machine in which the resource optimization manager is running (Machine A). The first step is to start generating the stress testing load imposed by the Open-STA tool. Then after 15 seconds from the test starting time, we start the whole property manager application, which starts up both the resource optimization manager, and security manager. The whole test lasts for almost 2 minutes. As the ROPM evaluates the incoming readings of the probing station sensor (shown as step 2), the probing station starts to probe the HTTP server (step3) and it returns the result back to the ROPM, but it first notifies the KB about its value change (step 4). Consequently the KB notifies all the registered subscribers according to their priority. Since the Security manager has a higher priority than the optimization manager, it receives the notification handle and starts evaluating the entire event based policies. After a period of 30 seconds the security manager releases the event to be handled by the next subscriber.

This experiment consists of two tests each presents a different scenario. Figure 4.31 illustrates the first test case used in this experiment where the security manager is not involved at all. In this test a load of 300 virtual users is generated and the apache tuning parameter value "ThreadsPerChild" is raised from 50 to 400. The response time plotted curve illustrates the improvement in the response time which took place after nearly 35 seconds of the test elapsed time due to restarting the apache server after increasing the value of the "ThreadsPerChild" apache tuning parameter. This action took place in accordance with the preconditions evaluation, which is defined in the HTTP policy file and monitored and executed by the resource optimization property manager.

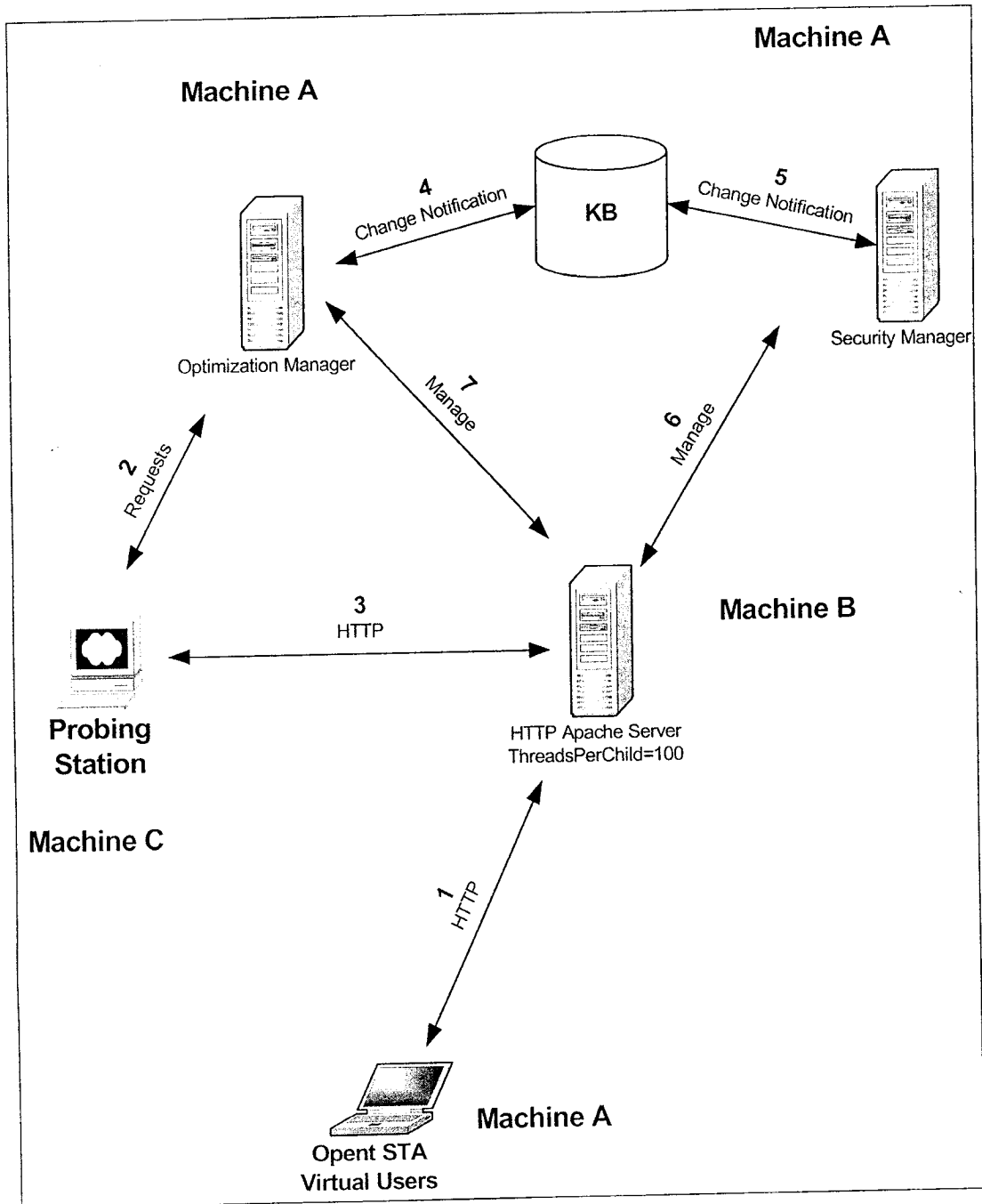


Figure 4. 30: Experiment Four Test Environment Setup

Figure 4.32 illustrates the second test case, which has the same settings as the first test in addition to the involvement of the security manager. For simplicity what the security manager actually does, is that it takes hold of the event notification by setting on the global lock for a delay of about 30 seconds and then it release it back. In figure 4.32 we notice a time shift in the response time improvement of almost 30 seconds relative to the improvement shown in figure 4.31, which is the delay time imposed by the security manager as it locks the global lock until it runs its simulated checks. Then the ROPM is able to hold the notification and take the corrective actions. We also notice the abrupt changes in the graph shown in Figure 4.32, which reaches the value of zero more than once. This behavior is actually due to two factors. The first one is the limitation of the used stress load generation tool (Open STA) in terms of the frame capture separation intervals, as the current version does some frame dropping during the performance snapshot which results in a sharp and abrupt change in the plotted curve and this of course means an acceptable errorpercentage in the graph. The second reason is due the tuning of the recorded test script itself, as it is not continuously producing an equivalent and homogenous load all the time. This produces a sharp edged and fluctuated response time curve as the one shown in most of the figures. This side effect could be minimized by, continuously tuning and adjusting the test scripts. In fact we did some minor test tuning and adjustment to minimize the mentioned effects. But we also kept in mind that our main purpose is to focus on illustrating the delay imposed by the security manager. Test tuning and optimization is a whole research area by itself, which is out of scope for this research.

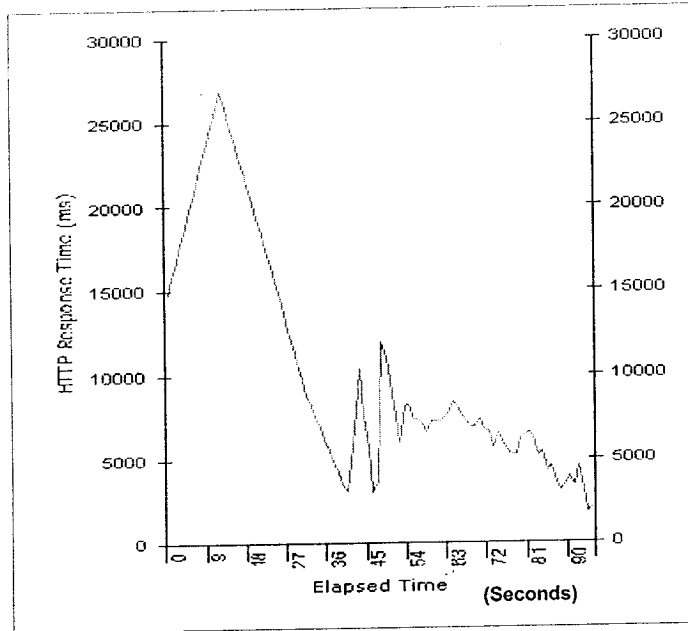


Figure 4. 31: Response Time For Test Case 1

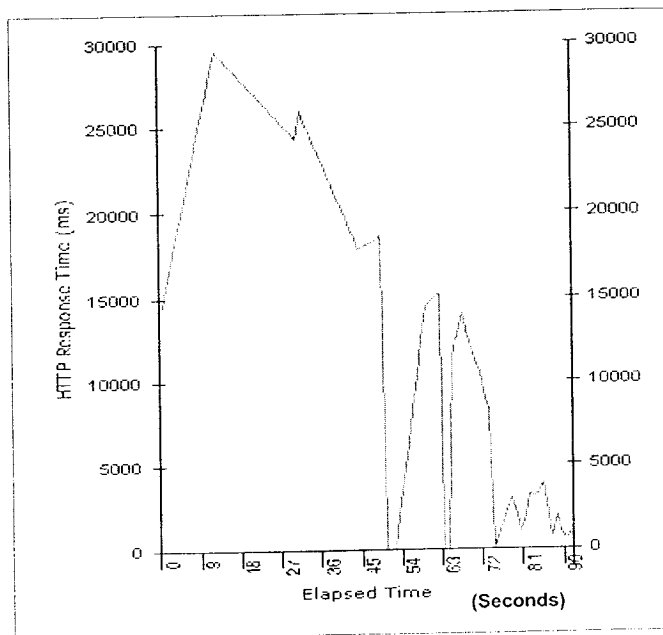


Figure 4. 32: Response Time For Test Case 2

4.6.3 Experiment Conclusion

In this experiment we have shown the feasibility of establishing a close cooperation between two type of autonomic property managers. This was demonstrated by the interrelationship between the resource optimization property manager and the security property manager. The experiment also illustrated the flexibility, simplicity, and reusability of any of the implemented sensors by more than one property manager at the same time by using the subscription/Notification mechanism provided that the systems with higher priority are notified before systems with lower priority. Once more, this model could be expanded to include all the different property managers, which can cooperate together and finally establish a self-management capable system, which is the foundation of an autonomic system.

4.7 Experiment Five

4.7.1 Experiment Objective.

The purpose of this experiment is to illustrate the benefits, and information can be obtained from the application generated log files. It also illustrates that by analyzing these results; the administrator can obtain some useful information about the efficiency of the parameters used in each policy in order to be able to tune them efficiently. In this experiment we used the same settings as experiment four, with some changes to the response time threshold values, as will be explained. All the given charts and analysis in this experiment are extracted from the application log file of which a snapshot is provided in figure 4.33. In the first test of this experiment we set the response time parameter for the website policy to 60 ms, which is a very small value that leads to more frequent calls to the assigned policy action. In this experiment specifically it means more frequent calls to the HTTP policy.

```

Invocation Type,TimeStamp,Policy Name,Function Name,Return value
Sensor,28-03-2004 21:46:36,websiteP,measuredValue,200
Sensor,28-03-2004 21:46:41,websiteP,measuredValue,181
Effector,28-03-2004 21:47:01,Security,FunctionCall,PolicyServer.SetKBVariable
Sensor,28-03-2004 21:47:28,HTTP,CPU_USAGE,9
Sensor,28-03-2004 21:47:29,HTTP,NumberOfProcess,90
Effector,28-03-2004 21:47:30,security,FunctionCall,PropertyManagerPack.Apacher
Effector,28-03-2004 21:47:30,websiteP,Policy,HTTP
Effector,28-03-2004 21:47:30,Security,FunctionCall,PolicyServer.SetKBVariable
Sensor,28-03-2004 21:47:31,websiteP,measuredValue,451
Sensor,28-03-2004 21:47:38,HTTP,CPU_USAGE,19
Sensor,28-03-2004 21:47:39,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:47:39,websiteP,Policy,HTTP
Sensor,28-03-2004 21:47:39,websiteP,measuredValue,70
Sensor,28-03-2004 21:47:46,HTTP,CPU_USAGE,20
Sensor,28-03-2004 21:47:46,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:47:46,websiteP,Policy,HTTP
Sensor,28-03-2004 21:47:47,websiteP,measuredValue,121
Sensor,28-03-2004 21:47:53,HTTP,CPU_USAGE,18
Sensor,28-03-2004 21:47:54,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:47:54,websiteP,Policy,HTTP
Sensor,28-03-2004 21:47:55,websiteP,measuredValue,70
Sensor,28-03-2004 21:48:00,HTTP,CPU_USAGE,16
Sensor,28-03-2004 21:48:01,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:48:01,websiteP,Policy,HTTP
Sensor,28-03-2004 21:48:02,websiteP,measuredValue,80
Sensor,28-03-2004 21:48:08,HTTP,CPU_USAGE,19
Sensor,28-03-2004 21:48:09,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:48:09,websiteP,Policy,HTTP
Sensor,28-03-2004 21:48:10,websiteP,measuredValue,70
Sensor,28-03-2004 21:48:16,HTTP,CPU_USAGE,24
Sensor,28-03-2004 21:48:16,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:48:16,websiteP,Policy,HTTP
Sensor,28-03-2004 21:48:17,websiteP,measuredValue,150
Sensor,28-03-2004 21:48:23,HTTP,CPU_USAGE,20
Sensor,28-03-2004 21:48:24,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:48:24,websiteP,Policy,HTTP
Sensor,28-03-2004 21:48:25,websiteP,measuredValue,80
Sensor,28-03-2004 21:48:30,HTTP,CPU_USAGE,23
Sensor,28-03-2004 21:48:31,HTTP,NumberOfProcess,400
Effector,28-03-2004 21:48:31,websiteP,Policy,HTTP

```

Figure 4.33: Application Log File

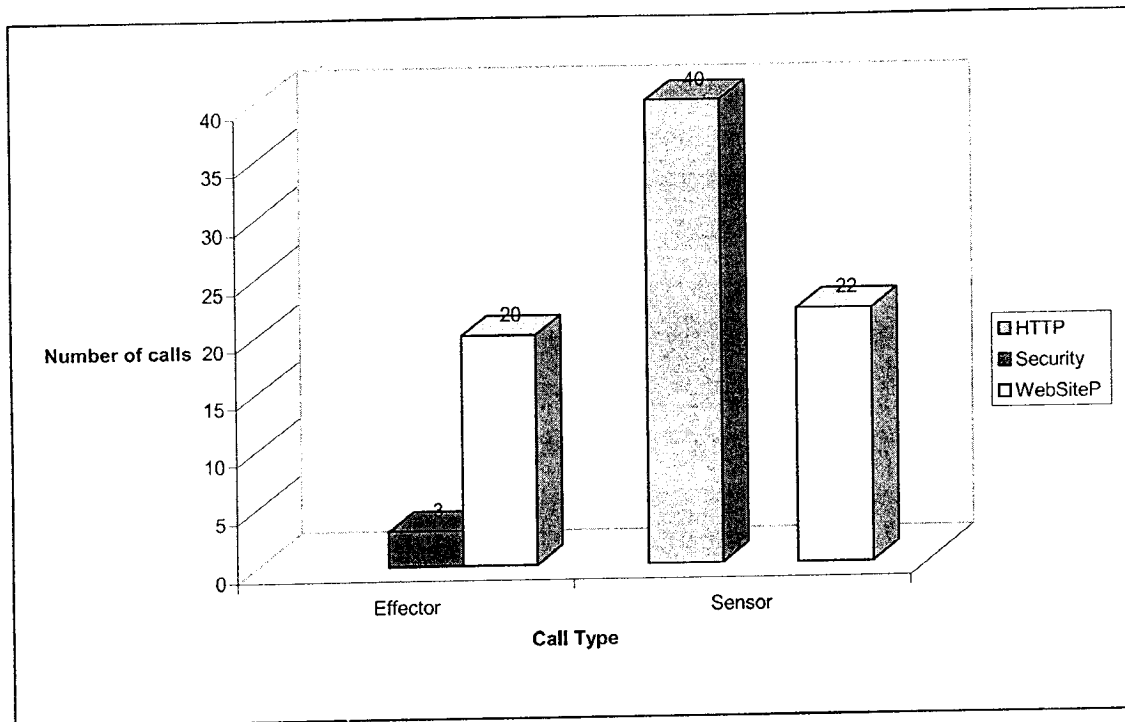


Figure 4. 34: Number of Calls Made Per Policy

4.7.2 Experiment Analysis

Figure 4.34 reflects the number of calls made for each policy. So in the graph we notice that for each 22 calls to the Website policy sensor, there are 40 calls to the HTTP policy sensor. Actually we have to divide the number of HTTP policy calls by the number of sensors in the precondition section for that policy. What actually happens is that each time a sensor inside the policy is evaluated, a transaction log is generated for that sensor call. So if we have an “IF” condition inside a policy that has more than one sensor operand “ANDED” “together, each call for each sensor generates a transaction in the log file to indicate the name of the invoked sensor and the invoking policy. Since we have two sensors in the HTTP policy, we have to divide the given number by two. Hence, we get a total of 20 calls for the HTTP policy, which almost means that each time the Website policy is fired, the HTTP policy is also called and that imposes an overhead on the Resource Property Manager without an actual benefit out of these calls. We can also notice that the security policy effector was invoked twice with no security sensor call! But we should remember that in fact the security

policy does not have any sensor, since it depends on the value returned by the Website policy sensor (the measured response time).

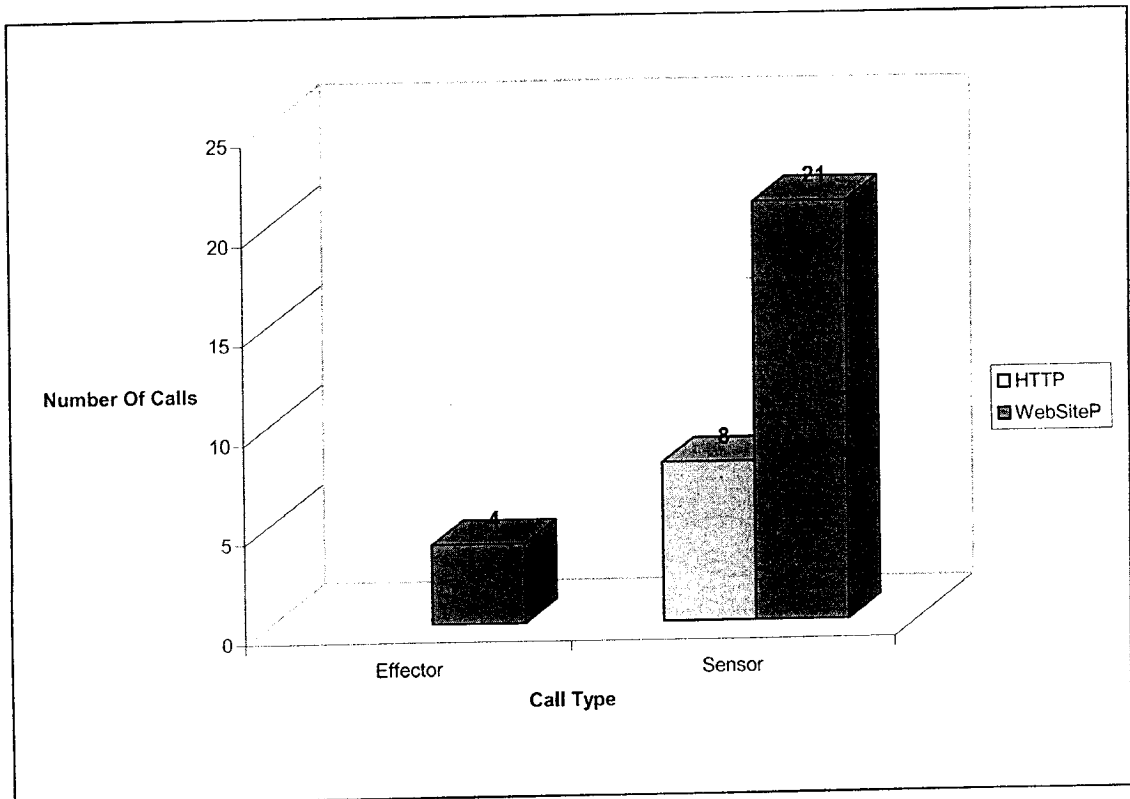


Figure 4.35: Number of Calls Made Per Policy In Test 2

In test 2 we increased the response time threshold up to 200 ms, and the outcome is depicted in figure 4.35, where the chart indicates a drop in the number of calls to the HTTP policy against the number of calls to the Website policy. We notice for the 24 visits of the Website policy that there are only 4 calls for the HTTP policy. So we can simply conclude that the used response time threshold value was not the optimum value, since it caused too many unnecessary policy invocations. This way we notice how simple it is to do some parameters tuning from the log files data analysis and charting.

Figure 4.36 depicts the plotted graph that can be generated from the log files to reflect different sensor readings during a snapshot of the test elapsed time. In this chart also the CPU

usage, measured response time, and the value of the apache tuning parameter “ThreadsPerChild” are all depicted. Such values and data analysis are useful in tracing the sensor data fluctuations during the test time, which helps in achieving policy tuning and improvement. For example the chart indicates that as we increased the “ThreadsPerChild” apache tuning parameter from 90 to 400 processes, we were able to enhance the response time as it dropped from 250 ms to an average of 100 ms. At the same time the value of the CPU utilization was not affected by that change and this indicates the efficiency of the used policy.

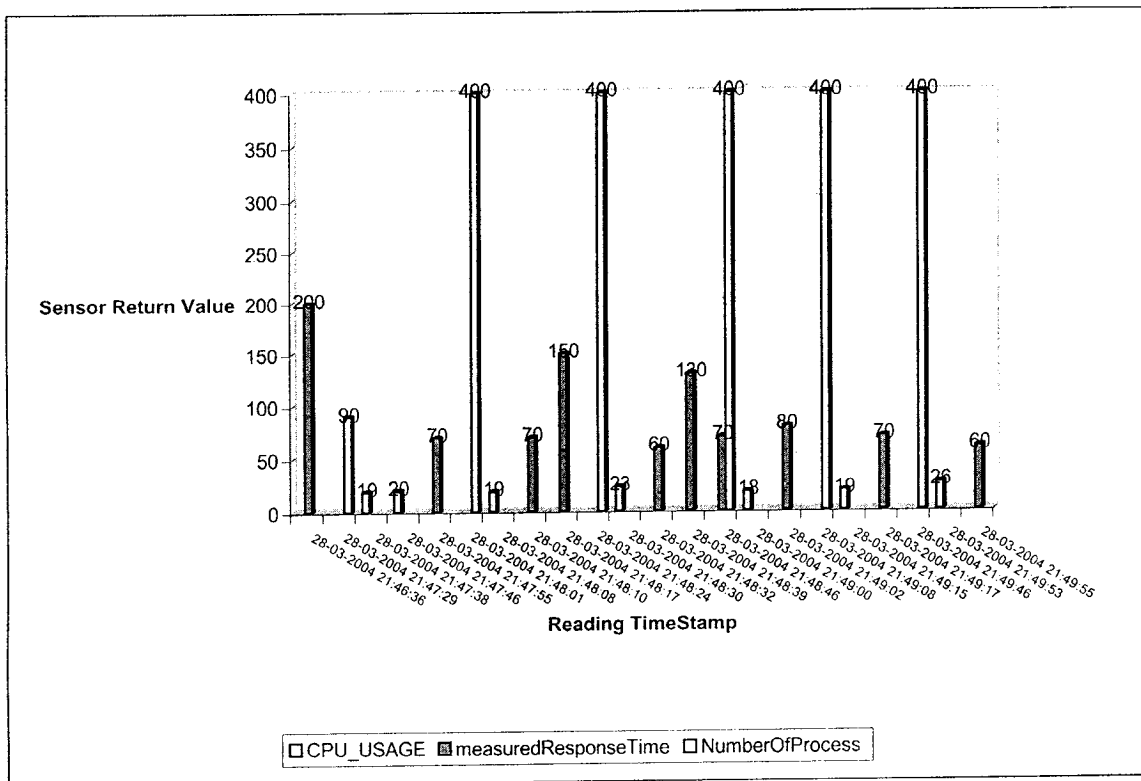


Figure 4.36: Sensor Readings During a Test Elapsed Time

4.7.3 Experiment conclusion

We demonstrated in this experiment some of the methods by which the generated logs can be used to enhance, tune, and analyze the written policies or even come out with more efficient policies that will smooth the work of the system administrator and consequently serve the business goal. The provided log analysis and charts in this experiment are just an example of the possible methods in which the log file data can be manipulated, whereas the system administrator is free to use whatever convenient method to gain the most out of the produced data. In this experiment we have simply used excel pivot tables and charts to analyze the log files data and come out with presented results. These data can be further loaded into a database server to give a more complex analysis and long-term tracking.

Chapter 5. Summary and Conclusion

5.1 Work Summary

In conclusion, this research work proposes a new notion to the autonomic computing architecture known as the property manager. The aim of the property manager is to represent each autonomic property separately by an autonomic manager that is capable of maintaining and handling the duties of the property it represents. This architecture provides the flexibility of adding any number of autonomic properties to the existing system without having to do any major modification to the system architecture. In addition, it provides us with a specialized type of autonomic managers that are goal oriented. For example, in our research work we presented the resource optimization property manager, which characterized the optimization autonomic property in terms of operation and duties. The resource optimization property manager handled the optimization management of a web server and an application server to illustrate the efficiency that it can provide to both systems when one specialized manager manages them. The resource optimization manager also demonstrated the capability of resource reallocation among the different systems that strive for consuming resources by distributing the available resources among the different systems with respect to a set of predefined policies and priorities that are defined according to the business needs. In our provided resource optimization manager we used the java language as the programming language and integrated the policy engine provided by IBM in the Autonomic Manager toolset (AMTS) package, which is a part of the ETTK [15], to generate our policies and rules that controlled the behavior of the optimization manager. In our experimental work we demonstrated the performance enhancement of the managed systems (Web Server and Application Server) before and after introducing the property manager to the system, and the flexibility that it introduces to the system by allowing more systems to join the environment easily. Finally we provided an example to illustrate the cooperation that can take place between the different property managers based on their priority definition on handling event notifications.

5.2 Contribution

The contributions made by this research work to the autonomic computing field are summarized in the following statements:

1. It presents a new perspective for the autonomic computing field through which we propose a solution that might lead to a flexible approach for dealing with the autonomic complexities and that makes use of the already existing software products and developed modules.
2. It presents a design model that decomposes the embedded complexity of any enterprise system into hierarchical layers to form together an autonomic system, which is easier to maintain and administrate.
3. It proposes solutions to the critical issues that the autonomic systems will have to handle such as the ability to provide goal specification and policies at the system management level for each property and to support high level goals at the business level.
4. It presents the new notion of the autonomic property manager to facilitate the management of any Enterprise environment autonomically by introducing a specialized manager in each autonomic property. The autonomic property manager monitors the managed systems and ensures the implementation of all the defined system policies for each system and provide services to any of the other property managers or autonomic managers according to its specialization.
5. It demonstrates how the property manager is made aware of both the systems and subsystems and can directly access and manage any of these subsystems when it needs to (e.g. take corrective actions or query the system).
6. It provides an implementation for the resource optimization property manager, which is capable of managing the resource optimization of more than one system concurrently each according to its predefined policies.

7. The presented model of the autonomic property manager was shown to help in the elimination of wrong problem tracking through the use of predefined system policies and separately specialized property managers (e.g. if there is an attack on a website the security manager starts working on the problem before letting the optimization manager investigates the performance issues).
8. Finally, It provides a high level definition for the business goals that are abstracted through the definition of hierarchical specialized policies associated with each subsystem (i.e. one can define policies for high level systems such as the corporate Website in addition to the ability of going deeper to describe subsystems' policies such as the Application Server, and the Web Server)

5.3 Problems faced and Concerns

Since autonomic computing is a new research area, it was very difficult to find a good deal of related research work and published papers. Additionally we had to review many different fields and technologies before we come out with our design model since the autonomic computing field includes more than one technology. Finally we faced great difficulties in using most of the related packages and software products due to the lack of documentation and poor support. We specially faced this problem with the AMTS policy engine [15] provided by IBM, as the documentation was very limited and insufficient. We had to post most of our questions through the only available forum provided by IBM but unfortunately we did not get any replies most of the times! In the design chapter of this work (and Appendix C) we mention the workarounds that we had to implement in order to overcome the unimplemented features in the used tools after we got the confirmation of the vendor that those features are not supported in the current version and that they will be provided in the near future. Most of these workarounds actually consumed lots of time and effort during the prototype implementation. In the remaining part of this section we mention some of the concerns, which might come to the reader's mind and are related to the architectural model used within this work in the form of questions and answers.

Q1: Are there any overheads going to be imposed on the managed systems due to the queries performed by the policy sensors?

A1: The overheads imposed by the sensors are most of the time of negligible value due to the fact that they are actually simulating the real load imposed by any normal system request. For example the sensor query that measures the website response time, is actually imposing an equal load to the one imposed by the normal user request. In addition, the hierarchical policy definition minimizes the need for invoking sensors unless needed. Actually if the system suffered from a performance problem at any point of time, the performance and response time of all the running systems will be affected and consequently the response time and performance of the autonomic property manager will be affected just as much as any running system within with the same environment until it takes a corrective action.

Q2: What about the security precautions that are taken during the client-server communication?

A2: We do not have much concern about the used security model and measurements since the whole system is never exposed to the outside world. It is always working within the enterprise network. However, the client server communication in our model is done through the JMS server, which provides its own security model that is based on the J2EE security model and that provides user authentication mechanism (Please refer back to the JMS specification for more details [29]).

Q3: What is the use of the autonomic property manager if the system is already managed by an autonomic manager isn't that a sort of redundancy?

A3: Property managers have a more global or let us say a wider view to the system as a whole and this relates to a higher perspective or comprehension of the business objectives and goals that are propagated to the lower systems (i.e. autonomic manager) in terms of specific commands. On the other hand the property manager could receive specific requests that are requested by the autonomic manager such as the request for resource allocation, which is a matter that cannot be handled by the autonomic manager. Additionally a property manager provides the advantage and knowledge of specialization in the form of services that can either be used by the policy author or the autonomic manager itself.

Q4: How can you guarantee that each property manager will only handle its own problems and will not interfere with other property managers?

A4: The prioritized event notification mechanism associated with each system policy definition guarantees that each property manager will only handle its related problems and according to the defined order through its priority.

5.4 Limitations and Directions for future work

The current system limitations and future work are very closely related issues since most of the system limitations are good materials for future work. Actually most of the limitations imposed on our model are either due to time or tool limitations. The AMTS policy engine [15] provided by IBM, which we used in our prototype implementation, is still in its early releases. The current version lacks many critical features that were mentioned in the system manuals and design description. Hence, as we expect the new release to come out soon, we hope it will contain those missing features, which will contribute positively to our model. Actually most of the future work is closely related to the future development of the autonomic management tools in general as this will also provide a much easier interface to the system management that can be integrated within the different autonomic models. However concerning our provided model we believe that the following points are good research points for future research as they are system limitations of the current model:

1. Expand the set of built-in functions provided by each property manager to be used by the systems administrator within his policy design in order to produce more efficient and practical policies (e.g. GetCpuUsage was the only implemented built-in function in our model).
2. Much more work has to be done regarding the inter- property manager communication (i.e. specifying the exact communication protocol between the different property managers).

3. More work has to be done regarding the global manager roles and responsibilities and the communication protocol between the global manager and the property managers.
4. More research has to be directed towards the validation of the implemented policies in terms of effectiveness and conflict resolution as the current design does not validate any of the provided policies before it is applied, however we provide a mechanism to track down the efficiency of the implemented policies.
5. Another important research area is related to the centralized control of the property managers and single points of failure. More work can be directed towards the ability of any of the property managers to startup another property manager to resume its duties when it fails (i.e. property manager recovery mechanism).
6. One of the current system limitation is the ability to roll back any of the corrective actions that are taken by one of the property managers. For example if the resource optimization manager did some optimization actions related to the http server it couldn't roll back these actions if it ever wanted to. Actually since the current system generates detailed log files they can easily be used to implement the rollback mechanism but it actually involves other important issues that are more complex than just setting back the old system values.
7. The command receiver module that resides on the client side to receive all the incoming messages and commands can only handle requests sequentially (i.e. one at a time). Actually this makes sense as we are using a JMS queue to handle all the in-coming requests but it also imposes some delay in the command execution and result return back during high traffic.
8. More work could be done to the GUI of the administration console to facilitate the job of the system administrator.
9. The rest of the property managers need to be implemented in order to provide the fully autonomic system that this research is heading for.
10. As the number of managed systems by the property manager increases, the environment will get more complicated and the management process will be more complex which might lead to a change in the shown levels of management

hierarchical structure. So it will be a good future research direction to validate the provided structure against a set of complex systems that are managed concurrently.

11. Finally, the Emerging Technologies Toolkit, which is provided by IBM, contains the AMTS tools in addition to other powerful toolsets, which present different technologies and techniques to deal with the autonomic computing. Some of these toolsets use AI based technologies, while others use Neural Network based technologies. Hence, future work can make use of these tools to extend the capabilities of the property manager and provide additional service such as the ability to provide a predictive reasoning for each policy, and constructing a problem solving knowledgebase.

Bibliography

- [1] "An architectural blueprint for autonomic computing", IBM Research Labs, April 2003.
- [2] A. G. Ganek, T. A. Corbi, "The dawning of the autonomic computing era", IBM SYSTEMS JOURNAL, VOL 42, NO. 1, 5-18 (2003).
- [3] ABLE: <http://www.research.ibm.com/able/>
- [4] Alexander Keller, Heiko Ludwig, "The WSLA Framerwork: Specifying and Mointoring Service Level Agreements for Web Services", Thomas J. Watson Research Center, May 2002.
- [5] Apache: <http://httpd.apache.org/docs/configuring.html>
- [6] Autonomic: <http://www.research.ibm.com/autonomic/overview>
- [7] Autonomic: <http://www.alphaworks.ibm.com/autonomic>
- [8] Bass, L; Clements, P.; and Kazman R. "Software Architecture in Practice." Boston, Ma.: Addison Wesley, March 1998.
- [9] BWLM: <http://alphaworks.ibm.com/tech/bwlm>
- [10] Catherine H. Crawford, Asit Dan, "e-model: Addressing the Need for a Flexible Modeling Framework in Autonomic Computing", IBM Research Division Thomas J. Watson Research Center, May 2002.
- [11] D.F. Bantz, C.Bisdikian, D. Challener, et al, "Autonomic Personal Computing" IBM Systems Journal 42, No. 1,165-176(2003).
- [12] Daniel H. Steinberg, " What you need to know about Autonomic Computing", developerWorks, August 2003.
- [13] David W. Levine, Ed Snible, Bill Arnold, et al, "Autonomic Toolset: Introduction To Autonomic Computing Toolset", IBM T.J. Watson Research Center Autonomic Toolset Project, July 2003.
- [14] David W. Levine, Ed Snible, Bill Arnold, et al, "Autonomic Toolset: Design Note", IBM T.J. Watson Research Center Autonomic Toolset Project, July 2003.
- [15] ETTK: <http://www.alphaworks.ibm.com/tech/ettk>
- [16] H. Ludwig, A. Keller, A. Dan, and R. King. "A service Level Agreement Language for dynamic electronic services". In WECWIS, June 26-28 2002.

- [17] H. Ludwig, A. Keller, A. Dan, R. Franck, and R.P. King. Web Service Level Agreement (WSLA) Language Specification. IBM Corporation, July 2002.
- [18] Hoi Chan, "Policy Usage in AMTS (Autonomic Manager Toolset)", IBM T.J. Watson Research Center Autonomic Toolset Project, July 2003.
- [19] IBM, "Autonomic Computing Concepts",
www3.ibm.com/autonomic/pdfs/AC_Concepts.pdf
- [20] IBM, "eWorkload Management Install and Configuration Guide", June 2003.
- [21] J2EE: <http://java.sun.com/j2ee/download.html>
- [22] J2SE: <http://java.sun.com/j2se/1.3/download.html>
- [23] JDeveloper: <http://otn.oracle.com/products/jdev/index.html>
- [24] J. P. Bigus, "The Agent Building and Learning Environment," Proceedings of the Fourth International Conference on Autonomous Agents (2000), pp. 108-109.
- [25] J. P. Bigus, D. A. Schlonsnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, "ABLE : A Toolkit for Building Multiagent Autonomic Systems", IBM Systems Journal 41, No. 3, 350-371 (2002).
- [26] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, "ABLE: A Toolkit for Building Multi-agent Autonomic Systems," IBM Systems Journal 41, No. 3, 350-371 (2002).
- [27] J. P. Bigus, J. L. Hellerstein, and M. S. Squillante, "Auto Tune: A Generic Agent for Automated Performance Tuning," Proceedings of the International Conference on Practical Application of Intelligent Agents and Multi-Agents (PAAM) (2000).
- [28] Jeffrey O.Kephart, Davi M.Chess, "The vision of Autonomic Computing" IEEE , January 2003.
- [29] JMS: <http://java.sun.com/products/jms/>
- [30] Kim Haase, "Java™ Message Service API Tutorial", Sun Microsystems Inc, 2002.
- [31] M. Agrawal, V. Bhat, H. Liu, et al, "AutoMate: Enabling Autonomic Applications on the Grid", Proceedings of Active Middleware Services (AMS), June 2003.
- [32] OpenSTA: <http://www.opensta.org/>
- [33] P. Horn, Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation (October 15, 2001); available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

- [34] S. Tuecke, K. Czajkowski, I. Foster, et al. "Grid service specification", <http://www.globus.org> Feb 2002.
- [35] Salim Hariri, C.S. Raghavendra, Yonhee Kim, Rinda P. Nelliudi, et al, "CATALINA: A Smart Application Control and Management", Active Middleware Services Conference, 2000.
- [36] Salim Hariri, Lizhi Xue, Huoping Chen, et al, "AUTONOMIA: An Autonomic Computing Environment ", Submitted to International Performance Computing and Communications Conference, 2003 (22nd IEEE).
- [37] WebSphere:<http://www106.ibm.com/developerworks/websphere/downloads/WASsupport.html>
- [38] WebSphere: <http://www-306.ibm.com/software/webservers/appserv/express/>
- [39] XML Schema Part 1: Structures. W3C Recommendation, W3 Consortium, May 2001.
- [40] XML Schema Part 2: Datatypes. W3C Recommendation, W3 Consortium, May 2001.
- [41] Y. Diao, J.L. Hellerstein, S. Parekh, and J.P Bigus, "Managing Web Server performance with AutoTune agents" IBM Systems Journal 42, No. 1,136-149(2003).

Appendix A: Glossary

Autonomic Nervous System

That part of the nervous system that governs involuntary body functions like respiration and heart rate.

Control Theory

The mathematical analysis of the systems and mechanisms for achieving a desired state under changing internal and external conditions. Cybernetics A term derived from the Greek word for "steersman" that was introduced in 1947 to describe the science of control and communication in animals and machines.

Feedback Control

A process by which output or behavior of a machine or system is used to change its operation in order to constantly reduce the difference between the output and a target value. A simple example is a thermostat that cycles a furnace or air conditioner on and off to maintain a fixed temperature.

GRID

Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements. The key distinction between clusters and grids is mainly lie in the way resources are managed. In case of clusters, the resource allocation is performed by a centralized resource manager and all nodes cooperatively work together as a single unified resource. In case of Grids, each node has its own resource manager and don't aim for providing a single system view

Grand Challenge

A problem that by virtue of its degree of difficulty and the importance of its solution, both from a technical and societal point of view, becomes a focus of interest to a specific scientific community. Grid computing A type of distributed computing in which a wide-ranging network connects multiple computers whose resources can then be shared by all end-users; includes what is often called "peer-to-peer" computing.

Artificial Intelligence (AI)

The capacity of a computer or system to perform tasks commonly associated with the higher intellectual processes characteristic of humans. AI can be seen as an attempt to model aspects of human thought on computers. Although certain aspects of AI will undoubtedly make contributions to autonomic computing, autonomic computing does not have as its primary objective the emulation of human thought.

Autonomic

- 1 Of, relating to, or controlled by the autonomic nervous system.
- 2 Acting or occurring involuntarily; automatic; an autonomic reflex.

Moore's Law

The observation made in 1965 by Gordon Moore, co-founder of Intel, that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. Moore predicted that this trend would continue for the foreseeable future. In subsequent years, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law, which Moore himself has blessed. Most experts, including Moore himself, expect Moore's Law to hold for at least another two decades.

Policy-based Management

A method of managing system behavior or resources by setting "policies" (often in the form of "if-then" rules) that the system interprets.

Quality of Service (QoS)

A term used in a Service Level Agreement (SLA) denoting a guaranteed level of performance (e.g., response times less than 1 second).

Service Level Agreement (SLA)

A contract in which a service provider agrees to deliver a minimum level of service.

Web Services Level Agreement (WSLA)

A framework that defines and monitors SLAs for Web Services.

Web Services

A way of providing computational capabilities using standard Internet protocols and architectural elements. For example, a database web service would use web browser interactions to retrieve and update data located remotely. Web services use UDDI to make their presence known.

Moore's Law

The observation made in 1965 by Gordon Moore, co-founder of Intel, that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. Moore predicted that this trend would continue for the foreseeable future. In subsequent years, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law, which Moore himself has blessed. Most experts, including Moore himself, expect Moore's Law to hold for at least another two decades.

Policy-based Management

A method of managing system behavior or resources by setting "policies" (often in the form of "if-then" rules) that the system interprets.

Quality of Service (QoS)

A term used in a Service Level Agreement (SLA) denoting a guaranteed level of performance (e.g., response times less than 1 second).

Service Level Agreement (SLA)

A contract in which a service provider agrees to deliver a minimum level of service.

Web Services Level Agreement (WSLA)

A framework that defines and monitors SLAs for Web Services.

Web Services

A way of providing computational capabilities using standard Internet protocols and architectural elements. For example, a database web service would use web browser interactions to retrieve and update data located remotely. Web services use UDDI to make their presence known.

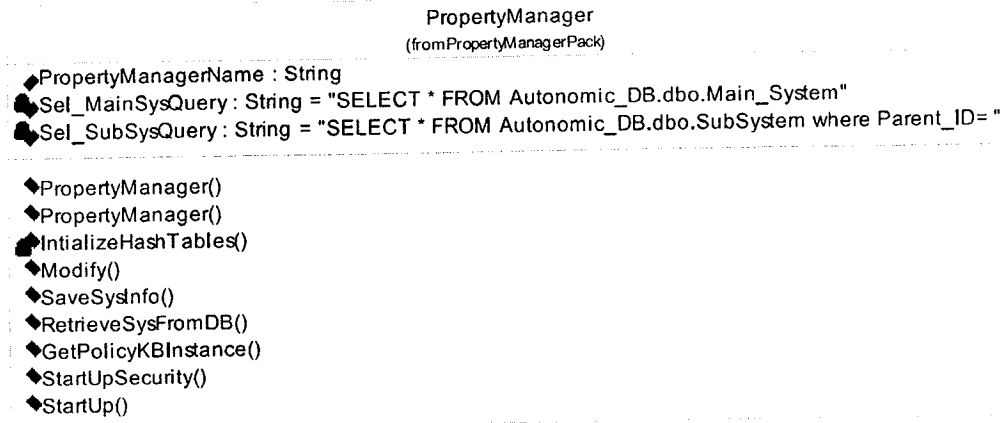
Appendix B: List Of Symbols

ABLE: Agent Building and learning environment
ADM: Application Delegated Manager
ADVICE: Adaptive Distributed Virtual Computing Environment
AE: Autonomic Element
AM: Autonomic Manager.
AME: Application Management Editor
AMS: Autonomic Middleware services
AMTS: Autonomic Manager Toolset
API: Application programming Interface
ARL: ABLE Rule Language
ARM: Application Response Measurement
BWLM: Business Workload Manager
ETTK: Emerging Technologies Toolkit
eWLM: e-workload management
GUI: Graphical User Interface
JMS: Java Messaging Service
KB: Knowledge Base
MIB: Management Information Base
OGSA: Open Grid Service Architecture
QoS: Quality of service
PTP: Point-To-Point.
ROI: Return of Investment
ROPM: Resource Optimization Property Manager
SLA: Service Level Agreement.
SNMP: Simple Network Management Protocol
TCO: Total Cost of Ownership
UDDI: Universal Description, Discovery, and Integration
WSLA: Web Services SLA
WSTK: Web Services Tool Kit

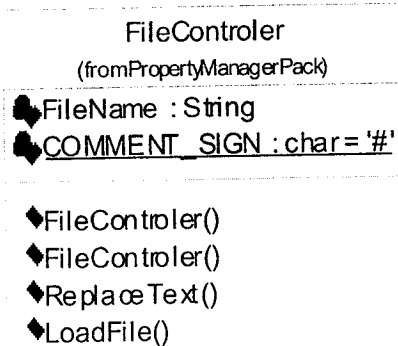
Appendix C: Workarounds

No.	Feature	Current AMTS Version	Workaround Solution	Example
1	The ability to trigger an episode of policy execution by another policy.	Current version support triggering an effector function only.	Passing the policy name to an effector function as a parameter in the normal policy file.	<ul style="list-style-type: none"> - SimplePolicyAction - policyName: "A" - MethodExpression - MethodName: "someMethod" - ReturnType: "void" - Parameter - Expression - Literal "Policy" - Literal - Expression - Expression - Literal "1111" - Literal - Expression
2	The ability to call a sensor function inside a consultative policy.	Only allowed in polling sensors, which are defined in event driven policies.	Map any sensor name of any type to a function call in the policy extension file.	<pre>method TestMeasuredFunctionAutonomicPackagePollingStationCarlSev's takes localhost, internal, misc IAW, handle names, false, ..</pre>
3	Parameterized sensor function call for any sensor type.	Current version support non-parameterized function calls for polling sensors only.	Map normal sensor name to a parameterized function call in the policy extension file.	<pre>method TestXCPI USAGP, FunctionPerfCounterCall, perf, size, 1000, ..</pre>

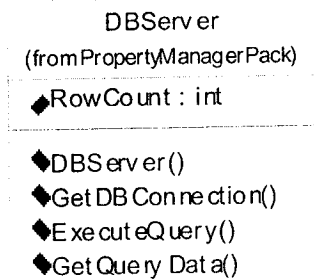
Appendix D: Detailed Class diagram



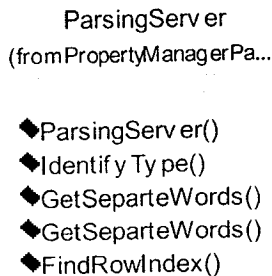
Property Manager Class



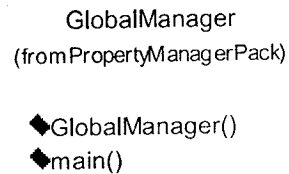
FileController Class



DBServer Class



ParsingServer Class



GlobalManager Class

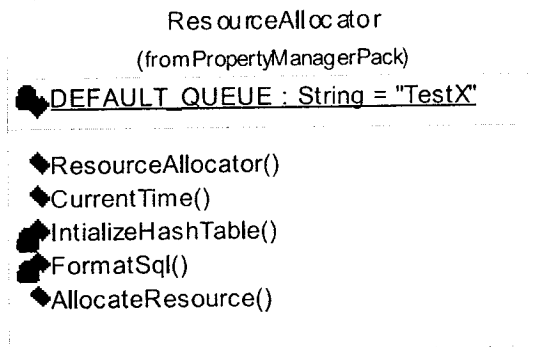
PolicyServer

(from PropertyManagerPack)

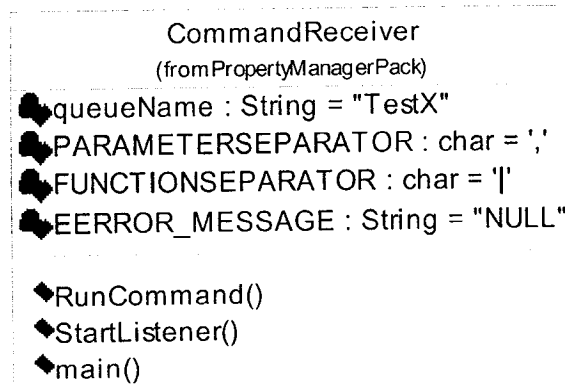
- ◆ KB_Current_Index : int
- ◆ KB_NUM : int = 5
- ◆ WORD_SEPARATOR : char = '.'
- ◆ MAX_FILE_CONTENT : int = 50
- ◆ SENSOR_KEYWORD : String = "Sensors:"
- ◆ EFFECTOR_KEYWORD : String = "Effectors:"
- ◆ SENSOR_FUNCTION_NAME : String = "PM_Sensor"
- ◆ DEFAULT_EFFECTOR_FUNC : String = "PM_Effector"
- ◆ DEFAULT_QUEUE : String = "TestX"
- ◆ ERROR_MESSAGE : String = "NULL"
- ◆ FUNCTIONSEPARATOR : char = '|'
- ◆ PARAMETERSEPARATOR : char = ','
- ◆ SENSOR_ARRAY_SIZE : int = 10
- ◆ EXPECTED_VALUE : int = 15
- ◆ PolicyName : String

- ◆ PolicyServer()
- ◆ SetKBVariable()
- ◆ GetPolicyHashTable()
- ◆ GetPolicyInstance()
- ◆ GetKBInstance()
- ◆ RunBuiltFunction()
- ◆ LoadPolicy()
- ◆ createKnowledgeBase()
- ◆ AddPolicyListener()
- ◆ LoadKnowledgeBase()
- ◆ createConsultativePolicy()
- ◆ createEventDrivenPolicy()
- ◆ eventNotification()
- ◆ LoadSensorEffectors()
- ◆ CmdClassifier()
- ◆ CPU_USAGE()
- ◆ measuredValue()
- ◆ PolicyInvoker()
- ◆ PM_Effector()
- ◆ StartPolicy()

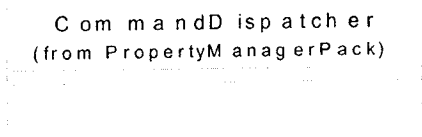
PolicyServer Class



ResourceAllocation Class



Command Receiver Class



CommandDispatcher Class

Appendix E: Code Samples for some important classes

1. PolicyServer.java

```
package PropertyManagerPack;

import java.io.InputStream;
import java.lang.reflect.Method;
import java.lang.Class;
import java.util.Collection;
import java.util.Collections;
import java.util.EventObject;
import java.util.Collections;
import java.util.Map;
import java.util.HashMap;
import java.util.List; // Remove if not used
import java.util.Hashtable;
import java.io.FileInputStream;
import java.util.Iterator;
import java.io.*;
import java.text.*;
import java.lang.*;

import com.ibm.autonomic.effector.DefaultEffector;
import com.ibm.autonomic.component.AutonomicException;
import com.ibm.autonomic.knowledge.AutonomicFact;
import com.ibm.autonomic.event.AutonomicEvent;
import com.ibm.autonomic.event.AutonomicEventListener;
import com.ibm.autonomic.component.AMTSLifecycleInit;
import com.ibm.autonomic.component.AMTSLifecycleStatus;
import com.ibm.autonomic.effector.Effector;
import com.ibm.autonomic.event.AutonomicMessageType; /***8
import com.ibm.autonomic.impl.knowledge.SimpleKnowledgeBase;
import com.ibm.autonomic.impl.policy.DefaultPolicyTranslator;
import com.ibm.autonomic.impl.policy.PolicyImpl;
import com.ibm.autonomic.impl.policy.policyEngine.AMTSPolicyEngineImpl;
import com.ibm.autonomic.impl.policy.resourceMapping.ResourceMappingImpl;
import com.ibm.autonomic.knowledge.Context;
import com.ibm.autonomic.knowledge.DefaultAutonomicFact;
import com.ibm.autonomic.knowledge.KnowledgeBase;
import com.ibm.autonomic.policy.Policy;
import com.ibm.autonomic.policy.ResourceMapping;
import com.ibm.autonomic.policy.Translator;
import com.ibm.autonomic.policy.policyEngine.AMTSPolicyEngine;
```

```
import com.ibm.autonomic.policy.policyEngine.ConsultativePolicy;
import com.ibm.autonomic.policy.policyEngine.EventDrivenPolicy;
import com.ibm.autonomic.policy.policyEngine.PolicyEngineException;
import com.ibm.autonomic.sensor.DefaultPollingAutonomicSensor;
import com.ibm.autonomic.sensor.PollingAutonomicSensor;
import com.ibm.autonomic.sensor.PassiveAutonomicSensor;
import com.ibm.autonomic.util.UnitOfTimeMeasure;

public class PolicyServer implements AutonomicEventListener//,Runnable
{
    private static KnowledgeBase[] kb;
    private static int KB_Current_Index;
    // private static ParsingServer PServer;
    private static Map PolicyHashTable;
    private static CommandDispatcher commandDispatcher;

    private EventDrivenPolicy Event_P;
    private ConsultativePolicy Consult_P;// it was static
    private Map SensorsFunTable;
    private Map TempSensorTable;
    private Map SensorNameToIndexMapping;
    private Map Function_IP_Mapping;
    private Map LocalFunTable;
    private Map NotificationTable;
    private Map SensorsTable;
    private Map EffectorsTable;
    private ParsingServer PServer;
    private Thread runner;
    private PollingAutonomicSensor[] valSensor;

    private final static int KB_NUM = 5;
    private final static char WORD_SEPARATOR = ',';
    private final static int MAX_FILE_CONTENT = 50;
    private final static String SENSOR_KEYWORD = "Sensors:";
    private final static String EFFECTOR_KEYWORD = "Effectors:";
    private final static String SENSOR_FUNCTION_NAME = "PM_Sensor";
    private final static String DEFAULT_EFFECTOR_FUNC = "PM_Effector";
    private final static String DEFAULT_QUEUE = "TestX";
    private final static String ERROR_MESSAGE = "NULL";
```

```

private final char FUNCTIONSEPARATOR='|';
private final char PARAMETERSEPARATOR =',';
private final static int SENSOR_ARRAY_SIZE = 10;
private final static int EXPECTED_VALUE = 15;

public String PolicyName;
/*****
/
public PolicyServer(String RecPolicyName)
{
    SensorsFunTable = Collections.synchronizedMap(new Hashtable());
    LocalFunTable = Collections.synchronizedMap(new Hashtable());
    NotificationTable = Collections.synchronizedMap(new Hashtable());
    SensorsTable = Collections.synchronizedMap(new Hashtable());
    EffectorsTable = Collections.synchronizedMap(new Hashtable());
    Function_IP_Mapping= Collections.synchronizedMap(new Hashtable());
    SensorNameToIndexMapping = Collections.synchronizedMap(new Hashtable());
    KB_Current_Index = 0; // Set the number of KB to Zero
    if (kb == null) kb= new SimpleKnowledgeBase[KB_NUM]; // Set KB Array to NULL

    if (commandDispatcher == null)
        commandDispatcher = new CommandDispatcher();

    if (PolicyHashTable == null)
        PolicyHashTable = Collections.synchronizedMap(new Hashtable());

    PolicyName = RecPolicyName;
    PolicyHashTable.put(RecPolicyName,this);

    //System.out.println("Constructor Intialized Class Parameters..!");
}

/*****/
public void SetKBVariable(String KeyName,Object OKeyValue)
{
    System.out.println("<<< .....SetKBVariable is Visited.....");
    try{
        Thread.sleep(20000); // This is the simulated sleeping time
    }catch(Exception e){System.out.println("Sleep Exception");}
    ParsingServer TmpPS = new ParsingServer();
}

```

```
String KeyValue = OKeyValue.toString();
if (TmpPS.IdentifyType(KeyValue) == 'I')
{
    kb[0].remove(KeyName,Context.GLOBAL_CONTEXT);
    kb[0].put(new DefaultAutonomicFact(KeyName,Context.GLOBAL_CONTEXT,
null, new Integer(Integer.parseInt(KeyValue))));
}
else if (TmpPS.IdentifyType(KeyValue) == 'F')
{
    kb[0].remove(KeyName,Context.GLOBAL_CONTEXT);
    kb[0].put(new DefaultAutonomicFact(KeyName,Context.GLOBAL_CONTEXT,
null, new Float(Float.parseFloat(KeyValue))));
}
else
{
    kb[0].remove(KeyName,Context.GLOBAL_CONTEXT);
    kb[0].put(new DefaultAutonomicFact(KeyName,Context.GLOBAL_CONTEXT,
null, new String(KeyValue)));
}
}
/*****/
public Map GetPolicyHashTable()
{
    return(PolicyHashTable);
}
/***** Get Policy Instance Function *****/
public Object GetPolicyInstance(String PolicyName)
{
    return(PolicyHashTable.get(PolicyName));
}

/*****/
public KnowledgeBase GetKBInstance(int KBIndex)
{
    if (KBIndex <= kb.length )
        return(kb[KBIndex]);
    else
        return(null);
}
/*****/
public String RunBUILTFunction(String FunctionText)
{

```



```
Method ReqMethod = null;
Object ReturnValue = null;
ParsingServer Parser = new ParsingServer();
String PackClass = null;
String MethodName = null;
Class RunX = null;
Object o = null;
String FuncParamList[] = null;
String DetailedCallList[] =
Parser.GetSeparateWords(FunctionText,FUNCTIONSEPARATOR);
try
{
    if(Integer.parseInt(DetailedCallList[0]) == 0)
    {
        return(EERROR_MESSAGE);
    }
    if(DetailedCallList[1].compareToIgnoreCase("BuiltIn") == 0)
    {
        PackClass = "PropertyManagerPack."+DetailedCallList[2]; //"AutonomicPackage."+
        Class CurrentClass = this.getClass();
        String CurrentClassName = CurrentClass.getName();
        CurrentClass.getMethods();
        MethodName = DetailedCallList[3];
        FuncParamList =
Parser.GetSeparateWords(DetailedCallList[4],PARAMETERSEPARATOR);
        if (PackClass.equalsIgnoreCase(CurrentClassName))
        {
            RunX = CurrentClass;
            o = this;
        }
        else
        {
            RunX = Class.forName(PackClass);
            o = RunX.newInstance();
        }
    }else // FOR OUTER IF
    {
        return("NoMatch");
    }
}
```

```
Method[] methods = RunX.getMethods();
for (int i=0; i<methods.length; i++)
{
    if (methods[i].getName().equals(MethodName))
    {
        ReqMethod = methods[i];
        break;
    }
} //FOR

Object TempObject[] = new Object[Integer.parseInt(FuncParamList[0])];
for (int i=1 ;i<=Integer.parseInt(FuncParamList[0]);i++)
{
    char Type = Parser.IdentifyType(FuncParamList[i]);
    if (Type == 'S')
        TempObject[i-1]= FuncParamList[i];
    else if (Type == 'I')
        TempObject[i-1] = new Integer(Integer.parseInt(FuncParamList[i]));
    else if (Type == 'F')
        TempObject[i-1] = new Float (Float.parseFloat(FuncParamList[i]));
    else if (Type == 'B')
        TempObject[i-1] = (Boolean.valueOf(FuncParamList[i]));
    else
        System.out.println("%%% Type is not Supported...");
} // for

ReturnValue = ReqMethod.invoke(o,TempObject);
if (ReturnValue == null )
    ReturnValue = "NULL";

System.out.println(">>> Return Values is "+ReturnValue.toString());
} catch (Exception e)
{
    System.out.println("%%% Exception Happened....");
    System.out.println(e.getMessage());
    return(EERROR_MESSAGE);
}

return(ReturnValue.toString());
} // FUNCTION
```

```

/*****
* Input : A List of Policy Files, Frequency of run of supplied function
*         The time unit of the supplied frequency
* Desc: This function upload the Customized Policy File and the other
* two policy files (main,mapping) by calling createPolicy fuction.
* @return Void
* @throws NoSuchMethodException
*****/
public void LoadPolicy(String[] PolicyFiles,String PolicyName,
                       int KB_Index,char PolicyType,int Freq,char TimeUnit)
{
    FileControler FC = new FileControler(PolicyFiles[2]);
    String[] FileContent = new String[MAX_FILE_CONTENT];
    try{
        // Load File
        FileContent = FC.LoadFile();

        // Create Parsing Server Instance
        if (PServer == null) PServer = new ParsingServer();

        //Load Sensor/Effector File with clear all flag
        LoadSensorEffectors(FileContent,true);

        // Create KB if it does not exist
        try{
            createKnowledgeBase(KB_Index);
        }catch (Exception e){System.out.println(e.getMessage());}

        // LoadKB KnowledgeBase
        LoadKnowledgeBase(KB_Index,PolicyType,Freq,TimeUnit);

        // Create a Policy with the kb
        String PolicyFile = PolicyFiles[0];
        String MappingFile= PolicyFiles[1];
        if ( PolicyType == 'E')
            Event_P =
createEventDrivenPolicy(kb[KB_Index],PolicyName,PolicyFile,MappingFile);
        else
            Consult_P =
createConsultativePolicy(kb[KB_Index],PolicyName,PolicyFile,MappingFile);
    }
}

```

```
catch (Exception e)
{
    System.out.println("%% Problem in LoadPolicy Function ...!");
    System.out.println(e.getMessage());
}
} // Function

/*****
 * Create a new Knowledge Base, insert the new created KB into the
 * array of The Class KB. IF -1 is indicated as the KB_Index a new
 * sequence will be generated however if a number is specified that
 * number will be used as the index of the created KB
 *
 * @return KnowledgeBase the created KnowledgeBase
 * @throws NoSuchMethodException
 *****/
private KnowledgeBase createKnowledgeBase(int KB_Index)
    throws NoSuchMethodException
{
    // create a SimpleKnowledgebase

    try
    {
        // -1 Index means that create next KB Index
        if (KB_Index == -1)
        {
            if (kb[KB_Current_Index] == null && KB_Current_Index <= KB_NUM)
                return(kb[++KB_Current_Index] = new SimpleKnowledgeBase());
        } // if
        else
        {
            if (kb[KB_Current_Index] == null && KB_Current_Index <= KB_NUM)
                return(kb[KB_Current_Index] = new SimpleKnowledgeBase());
        }
    } catch (Exception e)
    {
        System.out.println("%% Failed to Create a new kb : "+e.getMessage());
    }
    return(null);
}
```

```

} // Function
/*****
public void AddPolicyListener(String Key, KnowledgeBase KBase, PolicyServer
NotifiedPolicy, int Priority)
{
    if (KBase != null)
        KBase.addNameListener(Key, Context.GLOBAL_CONTEXT, false, this, Priority);
    else
    {
        Object SensorIndex = SensorNameToIndexMapping.get(Key);
        String TmpSensorVal = SensorIndex.toString();
        if (SensorIndex == null)
            System.out.println("*** Sensor Mapping Index Does not Exist...");
        else
            valSensor[Integer.parseInt(TmpSensorVal)].addNameListener(Key,
Context.GLOBAL_CONTEXT, false, NotifiedPolicy, Priority);
    }
}

/*****
* Load Knowledge Base, and put inside a sensor, a constant, and an effector.
* Start the sensor, which will provide a changing value. So there will be four
* things in the knowledge base.
* @return KnowledgeBase the created KnowledgeBase
* @throws NoSuchMethodException
*****/
private KnowledgeBase LoadKnowledgeBase(int Used_KB_Index, char PolicyType, int
Freq, char Unit)
    throws NoSuchMethodException
{
    // Declaration Section
    KnowledgeBase Used_KB = kb[Used_KB_Index];
    Method SensorMethod[] = new Method[SensorsTable.size()];
    valSensor = new PollingAutonomicSensor[SENSOR_ARRAY_SIZE];
        // Make a sensor which just calls getMeasuredValue()

    // ----- Normal Sensor Registration Section -----
    for (Iterator it = SensorsTable.keySet().iterator(); it.hasNext(); )

```

```

{
    String key = (String) it.next();
    String KeyValue = String.valueOf(SensorsTable.get(key));

    if (PServer.IdentifyType(KeyValue) == 'I')
        Used_KB.put(new DefaultAutonomicFact(key, Context.GLOBAL_CONTEXT, null,
new Integer(Integer.parseInt(KeyValue))));
    else if (PServer.IdentifyType(KeyValue) == 'F')
        Used_KB.put(new DefaultAutonomicFact(key, Context.GLOBAL_CONTEXT, null,
new Float(Float.parseFloat(KeyValue))));
    else
        Used_KB.put(new DefaultAutonomicFact(key, Context.GLOBAL_CONTEXT, null,
new String(KeyValue)));
} // For Loop

//----- Function/Polling Sensor Registration Section -----
int i = 0;
if (PolicyType == 'E')
{
    // Create a new polling sensor
    for (Iterator it = SensorsFunTable.keySet().iterator(); it.hasNext(); )
    {
        String key = (String) it.next();
        String KeyValue = String.valueOf(SensorsTable.get(key));

        // This is to get the right function within this class
        SensorMethod[i] = this.getClass().getMethod(key, null);
        try
        {
            SensorMethod[i], key, null);
            valSensor[i] = new DefaultPollingAutonomicSensor(key, this, SensorMethod[i], key,
null);
            SensorNameToIndexMapping.put(key, new Integer(i));
        } catch (Exception e) { System.out.println(e.toString()); }

        // Tell KB about sensor
        valSensor[i].register(Used_KB);

        /* Tell sensor to publish into knowledge base
        * Actually this statement will make the sensor (valSensor) make a call
        * to the KB each time the value of the sensorName changes

```

```
    * A larger number means higher priority in listening events!!
    **/
    // This enable the KB to call event notification
        valSensor[i].addNameListener(key, Context.GLOBAL_CONTEXT, false,
this, 0);

    // This Statement notifies the KB and hence triggers the Policy
    valSensor[i].addNameListener(key, Context.GLOBAL_CONTEXT, false, Used_KB,
7);
        // Start sensor
        valSensor[i].setPollingRate(Freq, UnitOfTimeMeasure.Minutes); // 10 times per
minute

        if (!valSensor[i].start(AMTSLifecycleInit.INITIAL_LOAD)) {
            System.out.println("%%% Couldn't start sensor "+key);
        };

    // This is to test how to capture an event as it calls the local function
    // kb.addNameListener("measuredValue", Context.GLOBAL_CONTEXT, false, p, 0);

        i++;
    } // For loop
} // IF for Policy Type
//----- Notification Registration Section -----
for (Iterator it = NotificationTable.keySet().iterator(); it.hasNext(); )
{
    String key = (String) it.next();
    String KeyValue = String.valueOf(NotificationTable.get(key));
    // Scar for the desired policy first
    ParsingServer TmpParseServ = new ParsingServer();
    String [] TmpFuncParamList =
TmpParseServ.GetSeperateWords(KeyValue,PARAMETERSEPARATOR);

    if (TmpFuncParamList[1].equalsIgnoreCase("SensorListener" )
    {
        PolicyServer TmpPolicy = (PolicyServer) GetPolicyInstance(TmpFuncParamList[2]);
        TmpPolicy.AddPolicyListener(key,null,this,Integer.parseInt(TmpFuncParamList[3]));
    }
    else
        AddPolicyListener(key,Used_KB,this,Integer.parseInt(TmpFuncParamList[3]));
}
```

```
//Used_KB.addNameListener(key,Context.GLOBAL_CONTEXT,false,this,Integer.parseInt(
KeyValue));

    }// For Loop

//----- Effector Registration Section -----

        Method EffectorMethod=
this.getClass().getMethod(DEFAULT_EFFECTOR_FUNC, new Class[]
{Object.class,Object.class});

    Iterator it = EffectorsTable.keySet().iterator();
    String key = (String) it.next();
    String KeyValue = String.valueOf(EffectorsTable.get(key));

    Effector eff = new DefaultEffector(key, this, EffectorMethod, KeyValue);

        // Register the effector in the knowledge base
    eff.register(Used_KB);

        return Used_KB;
    }// Function

/*****
****
* Create a consultative policy instance which is connected to a knowledge base.
* @param kb the knowledgeBase to be connected by the event driven policy instance
* @return ConsultativePolicy the ConsultativePolicy created
****
*****/

private ConsultativePolicy createConsultativePolicy(KnowledgeBase kb,String
PolicyName,String PolicyFile,String MappingFile)
        throws PolicyEngineException, PolicyEngineException,
FileNotFoundException
{
        // Read a policy from the classpath

        // Read a policy from the classpath
```



```
InputStream is =(InputStream) new FileInputStream(PolicyFile);
InputStream is2 =(InputStream) new FileInputStream(MappingFile);

    // Construct a policy
        Policy policy = new PolicyImpl(is);
ResourceMapping mapping = null;
    try {

mapping = new ResourceMappingImpl(is2);

    }
catch (Exception e)
{
        System.out.println(e.toString());
    }

    // Create a DefaultPolicyTranslator
Translator trans = new DefaultPolicyTranslator();

// Create an instance of default PolicyEngine
    AMTSPolicyEngine engine = new AMTSPolicyEngineImpl();
ConsultativePolicy cpolicy = null; // added for testing
try{
        // Create an instance of the Consultative policy

cpolicy = engine.createConsultativePolicyInstance(
                policy,
                trans,
                mapping,
                PolicyName,
                kb);}
catch (PolicyEngineException e)
{
    System.out.println(e.getMessage());
}

    return cpolicy;

}

/*****
```

```

* Create an event-driven policy instance which is connected to a knowledge base.
* @param kb the knowledgeBase to be connected by the event driven policy instance
* @return ConsultativePolicy the ConsultativePolicy created
*****/
private EventDrivenPolicy createEventDrivenPolicy(KnowledgeBase kb,String
PolicyName,String PolicyFile,String MappingFile)
throws PolicyEngineException, PolicyEngineException,
FileNotFoundException
{
    // Read a policy from the classpath
    InputStream is =(InputStream) new FileInputStream(PolicyFile);
    InputStream is2 =(InputStream) new FileInputStream(MappingFile);

// Construct a policy
    Policy policy = new PolicyImpl(is);
    ResourceMapping mapping = null;
    try {

        mapping = new ResourceMappingImpl(is2);

    }
    catch (Exception e)
    {
        System.out.println(e.toString());
    }

    // Create a DefaultPolicyTranslator
    Translator trans = new DefaultPolicyTranslator();

// Create an instance of default PolicyEngine
    AMTSPolicyEngine engine = new AMTSPolicyEngineImpl();

    EventDrivenPolicy epolicy = engine.createEventDrivenPolicyInstance(
        policy,
        trans,
        mapping,
        PolicyName,
        kb);

    return epolicy;
} // End Create Policy

/*****

```

```

* (Called when an event arrives.)
*
* @param inboundEvent an autonomic event
*****/
public AutonomicEvent eventNotification(AutonomicEvent ev) throws AutonomicException
{
    AutonomicFact fact = ev.getFact();

    System.out.println("I got the event "+this.PolicyName);
    return (ev);
}

/***** LoadSensorEffectors Function *****/
* Input    : Array of Strings.
* Description : Actually in this function we load three different types of
* sensors to three tables :-
* SensorsFunTable : This hash table will contain all external fun calls
* SensorsTable    : This hash table will contain all var to be stored in the kb
* LocalFunTable   : This hash table will contain all local fun calls
* EffectorsTable  : For the effecotrs we use one hash table only.
* @return : Void.
*****/
private void LoadSensorEffectors(String[] FileContent,boolean ClearAll)
{
    if (ClearAll)
    {
        SensorsFunTable.clear();
        SensorsTable.clear();
        LocalFunTable.clear();
        NotificationTable.clear();
        EffectorsTable.clear();
    }
    //Create an instance of the parser class
    ParsingServer ParsSrv = new ParsingServer();
    String Words[] ;
    // Locate Sensors: Keyword
    int Sensorindex = ParsSrv.FindRowIndex(FileContent,SENSOR_KEYWORD);
    // Locate Effector Keyword
    int Effectorindex = ParsSrv.FindRowIndex(FileContent,EFFECTOR_KEYWORD);

    // Need to be raised as an exception later
    if (Sensorindex == -1) System.out.println("%%% No Sensor Section Found !");
}

```

```
// Loop on Sensors to handle them
for(int i=Sensorindex+1;i<Effectorindex;i++)
{
// IF First Key is method then WORD_SEPARATOR '='
Words = ParsSrv.GetSeparateWords(FileContent[i],WORD_SEPARATOR);
if (Words[1].compareToIgnoreCase("Method")==0 )
{
try{
/* Insert the name of the sensor first ,
* then the function name and its parameters
*/
SensorsFunTable.put(Words[3],new String(Words[4]));
Function_IP_Mapping.put(Words[3],Words[2]);
}
catch(Exception e){System.out.println(e.getMessage());}
}
// IF First Field indicates Value
else if(Words[1].compareToIgnoreCase("Value")==0 )
{
SensorsTable.put(Words[2],new String(Words[3]));
// IF First Field indicates a Local function call
}else if(Words[1].compareToIgnoreCase("local")==0 )
{
LocalFunTable.put(Words[2],new String(Words[3]));
//Function_IP_Mapping.put(Words[3],Words[2]);
}else if(Words[1].compareToIgnoreCase("Notify")==0 )
{
NotificationTable.put(Words[2],new String(Words[3]));
}else
System.out.println("%%% No Such Token Syntax : "+Words[0]);

} // FOR LOOP

// ----- Loop on Effectors to handle them -----
int TotalLineNum = Integer.parseInt(FileContent[0]);
for(int i=Effectorindex+1;i<=TotalLineNum;i++)
{
Words = ParsSrv.GetSeparateWords(FileContent[i],WORD_SEPARATOR);
if (Words[1].compareToIgnoreCase("Method")==0 )
{
EffectorsTable.put(Words[3],new String(Words[4]));
}else
```

```

        System.out.println("%% No Such Token Syntax : "+Words[1]);
    }// FOR LOOP

} // Function

/***** Sensor Functions *****/
* Input    : CmdClassifier.
* Description : This function takes a Key Function name and
* hash its equivalent value from one table and gets its Queue/IP name
* then send these info to the Command Dispatcher in order to execute
* it
*
* @return: the result returned back from the command dispatcher.
*****/
protected String CmdClassifier(String KeyName,boolean IsEffector)
{
    Object KeyValue =null;
    String Result =null;
    // We should do hashing on the function name
    if (IsEffector)
        Result = commandDispatcher.ExecuteCall(DEFAULT_QUEUE,KeyName.toString());

    else
    {
        KeyValue = SensorsFunTable.get(KeyName); //"measuredValue"
        Object QueueName = Function_IP_Mapping.get(KeyName);
        Result =
commandDispatcher.ExecuteCall(QueueName.toString(),KeyValue.toString());
    }

    // System.out.println("*****Returned Key Value is: "+KeyValue.toString());

    return(Result);
}
/***** Sensor Functions *****/
* Input    : Null.
* Description :
*
* @return:

```

```
*****/
public int CPU_USAGE()

{
    System.out.println(">>> CPU_USAGE Property Manager Sensor Triggered: ");

    int iResult=0;

    String Result =CmdClassifier("CPU_USAGE",false);
    if (Result.compareToIgnoreCase("Null")==0)
        iResult=0;
    else
        iResult = Integer.parseInt(Result);
    return(iResult);
}
/***** Sensor Functions *****/
* Input      : Null.
* Description :
*
* @return:
*****/
public int measuredValue()
// PM_Sensor()
{
    System.out.println(">>> MeasuredValue Property Manager Sensor Triggered: ");

    int iResult=0;

    String Result =CmdClassifier("measuredValue",false);
    if (Result.compareToIgnoreCase("Null")==0)
        iResult=0;
    else
        iResult = Integer.parseInt(Result);
    return(iResult);
}

/***** Effector Function *****/
*
*
*
*****/
private void PolicyInvoker(String PolicyName)
```

```
{
    try
    {
        char[] x={'C'};
        Object PolicyInstance;
        Object[] FunctionParameter = new Object[3]; //{PolicyName};
        FunctionParameter[0] =PolicyName;
        FunctionParameter[1] = new Integer(0);
        FunctionParameter[2] = new String(x);

        PolicyInstance = PolicyHashTable.get(PolicyName);
        Class PolicyClass = PolicyInstance.getClass();
        Method[] methods = PolicyClass.getMethods();
        Method ReqMethod= null;

        for (int i=0; i<methods.length; i++)
        {
            if (methods[i].getName().equals("StartPolicy"))
            {
                ReqMethod = methods[i];
                break;
            }
        }

        ReqMethod.invoke(PolicyInstance,FunctionParameter);
        System.out.println(">>> Fired Policy : "+PolicyName);
    }//try
    catch(Exception e){}

    }// Function
    /*****/
    public void PM_Effector(Object Type,Object Message)
    {

        System.out.print(">>> Property Manager Effector Triggered msg Type :");
        System.out.print(Type);
        System.out.print(" Message value: ");
        System.out.println(Message);
        String PolicyName = (String) Message;
        String PolicyType = (String) Type;
        String EffectorReturnValue=null;
    }
}
```

```

if(PolicyType.compareToIgnoreCase("Policy")== 0)
    PolicyInvoker(PolicyName);
else if(PolicyType.compareToIgnoreCase("FunctionCall")== 0)
    {
        EffectorReturnrValue = RunBuiltFunction(PolicyName);
        if (EffectorReturnrValue.equalsIgnoreCase("NoMatch"))
            EffectorReturnrValue= CmdClassifier(PolicyName,true);
    }

    System.out.println(">>> Effector Returned : "+EffectorReturnrValue);

}

/***** StartPolicy Function *****/
* Input      :
* Description :
* 'E' here means that it is an event driven policy.
* @return:
*****/
public void StartPolicy(String SensorName,int KB_Index,String PolicyType)
        throws NoSuchMethodException, PolicyEngineException,
        InterruptedException
    {
        try
        {

            TempSensorTable = new HashMap();
            TempSensorTable.clear();

            if (PolicyType.compareToIgnoreCase("E") == 0)
            {
                if (!Event_P.start(AMTSLifecycleInit.INITIAL_LOAD))
                    System.out.println("%%% Couldn't start sensor in StartPolicy");
            }else
            {
                System.out.println(">>> System Policy Will be Consulted..");

                ParsingServer ParseServerInst = new ParsingServer();

                for (Iterator it = this.SensorsFunTable.keySet().iterator(); it.hasNext(); )

```



```
{
    String key = (String) it.next();
    String KeyValue = String.valueOf(this.SensorsFunTable.get(key));
    String Result = CmdClassifier(key,false);

    AutonomicFact AF = this.kb[KB_Index].get(key);
    List XString = AF.getFactDataAsList();
    if (Result.equalsIgnoreCase("null"))
    {
        System.out.println("*** ERROR: System Value ["+key+"] Could not be updated
(NULL) !");
    }
    else if (ParseServerInst.IdentifyType(Result) == 'I')
    {
        kb[KB_Index].remove(key,Context.GLOBAL_CONTEXT);
        kb[KB_Index].put(new DefaultAutonomicFact(key,Context.GLOBAL_CONTEXT,
null, new Integer(Integer.parseInt(Result))));
    }
    else if (PServer.IdentifyType(Result) == 'F')
    {
        kb[KB_Index].remove(key,Context.GLOBAL_CONTEXT);
        kb[KB_Index].put(new DefaultAutonomicFact(key,Context.GLOBAL_CONTEXT,
null, new Float(Float.parseFloat(Result))));
    }
    else
    {
        kb[KB_Index].remove(key,Context.GLOBAL_CONTEXT);
        kb[KB_Index].put(new DefaultAutonomicFact(key,Context.GLOBAL_CONTEXT,
null, new String(Result)));
    }

} // FOR

Consult_P.consultPolicy(Context.GLOBAL_CONTEXT);
} // ELSE

//AMTSLifecycleStatus IC = p.getStatus();
} // try
catch (Exception e)
{
```

```
System.out.println(" Exception in StartPolicy Section :");
System.out.println(e.getMessage());
}
/* for (;) { // forever
// Sleep for 1 second
Thread.sleep(1000);
} // FOR*/

} // StartPolicy
} // CLASS
```

2. ResourceAllocator.java

```
package PropertyManagerPack;

import java.sql.*;
import java.util.*;
import java.sql.Timestamp;

public class ResourceAllocator
{
    private static DBServer dbServ;
    private static CommandDispatcher commandDispatcher;
    private final static String DEFAULT_QUEUE = "TestX";
    private Map RSTable;

    /**
     * Input :
     * @return void
     * @throws
     */
    public ResourceAllocator()
    {
        if (dbServ == null )
            dbServ = new DBServer();// Creating DB Server Instance
    }
}
```

```
if( commandDispatcher == null)
    commandDispatcher = new CommandDispatcher();
// Establish DB Connection and run query
dbServ.GetDBConnection("ULTRA","sa");
RSTable = Collections.synchronizedMap(new Hashtable());
}

/*****
 * Input :
 * @return void
 * @throws
 *****/

static public String CurrentTime()
{
    Calendar cal = Calendar.getInstance(TimeZone.getDefault());
    String DATE_FORMAT = "dd-MM-yyyy HH:mm:ss";
    java.text.SimpleDateFormat sdf = new java.text.SimpleDateFormat(DATE_FORMAT);
    sdf.setTimeZone(TimeZone.getDefault());
    return(sdf.format(cal.getTime()));
}

/*****
 * Input :
 * @return void
 * @throws
 *****/

private void IntializeHashTable(Map RSTable)
{
    RSTable.put("IP","");
    RSTable.put("ResourceAcqCMD","");
    RSTable.put("OwnerSysID","");
    RSTable.put("ResourceID","");
    RSTable.put("ResourceQuery","");
    RSTable.put("CurrentCost","");
    RSTable.put("LastQueryUpdate","");
}

/*****
 * Input :
 * @return void
 *****/
```

```

* @throws
*****/
private String FormatSql(String QueryName,String[] Parameters)
{
    String SQLStmt="No Query";
    String SQLStmtPart0 = " SELECT ";
    String SQLStmtPart1 = "RC.OwnerSysID, RC.ResourceID, RC.ResourceQuery,
RC.LastQueryUpdate, RC.CurrentCost";
    String SQLStmtPart2= " FROM Autonomic_DB.dbo.AUT_Main_System MS INNER
JOIN ";
    String SQLStmtPart3 = " Autonomic_DB.dbo.AUT_ResourceConsumption RC ON MS.ID
= RC.OwnerSysID";
    String SQLStmtPart4 = "";
    String SQLStmtPart5 = "";
    String SQLStmtPart6 = "";
    String SQLStmtPart7 = " AND (MS.parent_id <>0)";
    String SQLStmtPart8 = " AND (RC.OwnerSysID NOT IN (SELECT ownersysid FROM
Autonomic_DB.dbo.V_AUT_PeakTimes))";
    String SQLStmtPart9 = " GROUP BY RC.OwnerSysID, RC.ResourceID,
RC.ResourceQuery, RC.LastQueryUpdate, RC.CurrentCost";
    String SQLStmtPart10= " ORDER BY MS.Priority DESC, RC.CurrentCost";
    SQLStmtPart5 = " WHERE (MS.Priority > "+Parameters[0]+") ";
    SQLStmtPart6 = " AND (RC.ResourceType ="+Parameters[1] +")";

    if (QueryName.equalsIgnoreCase("InsertTransQuery"))
    {
        String TmpSQL = "INSERT INTO Autonomic_DB.dbo.AUT_ResourceTransaction"
            +"(DonorSysID, ConsumerSysID, ResourceName, TransactionTime)"
            +"VALUES("+Parameters[0]+","+Parameters[1]
            +" , "+Parameters[2]+","+Parameters[3]+")";
        return(TmpSQL);
    }
    else if (QueryName.equalsIgnoreCase("UpdateResourceQuery"))
    {
        String TmpSQL1 = "update Autonomic_DB.dbo.AUT_ResourceConsumption set
LastQueryUpdate = ";
        String TmpSQL2= " , CurrentCost = ";
        String TmpSQL3= "where OwnerSysID= ";
        String TmpSQL4=" and ResourceID = ";
        String FinalSQL= TmpSQL1+Parameters[0]+""+TmpSQL2+Parameters[1]+""
            +TmpSQL3+Parameters[2]
            +TmpSQL4+Parameters[3];
    }
}

```

```

    return(FinalSQL);
} else if (QueryName.equalsIgnoreCase("ResourceQuery"))
{
    SQLStmt= SQLStmtPart0+ SQLStmtPart1 +",MS.IP "+ SQLStmtPart2 +
        SQLStmtPart3 + SQLStmtPart4 +
        SQLStmtPart5 + SQLStmtPart6+
        SQLStmtPart7 + SQLStmtPart8+SQLStmtPart9+",MS.IP";

} else if (QueryName.equalsIgnoreCase("ResAcqQuery"))
{
    SQLStmtPart1 = "MS.Priority, RC.OwnerSysID, RC.ResourceID, RC.ResourceQuery,
RC.ResourceAcqCMD, RC.LastQueryUpdate, RC.CurrentCost";
    SQLStmtPart9 = " GROUP BY MS.Priority, RC.OwnerSysID, RC.ResourceID,
RC.ResourceQuery, RC.ResourceAcqCMD, RC.LastQueryUpdate, RC.CurrentCost";

    SQLStmt= SQLStmtPart0+ SQLStmtPart1 + SQLStmtPart2 +
        SQLStmtPart3 + SQLStmtPart4 +
        SQLStmtPart5 + SQLStmtPart6 +
        SQLStmtPart7 + SQLStmtPart8 +
        SQLStmtPart9 + SQLStmtPart10;

}
return(SQLStmt);
}

/*****
* Input : A string that contains the supplied ResourceType (e.g CPU)
* and another string that contains the Requesting system name (e.g. Website)
* Desc: This function does a search for all the running systems that
* are using the same supplied resource and that has a lower priority
* and that are not currently working on their peek time.
* @return void
* @throws SQL Exception
*****/
public int AllocateResource(String ResourceType , String RequesterName,boolean Commit)
{
    String SQL.Stmt =null;
    String [] ParameterList =new String[5];
    String SysIP =null;
    System.out.println(">>> ResourceAllocator is visited...");

```

```
try
{
    String PrimarySQLStmt = "SELECT ID, Priority, IP FROM
Autonomic_DB.dbo.AUT_Main_System MS WHERE (Sys_Name =
"+RequesterName+"");
    ResultSet Main_RS = dbServ.ExecuteQuery(PrimarySQLStmt,false);

    /* We assume that systems with higher priority number are less important
    * Systems. (I.E. Priority 1 is the highest). We also assume that the
    * main systems (i.e. the ones with parent Zero are excluded from the query
    */

    if (Main_RS.next())
    {
        ParameterList[0] = Main_RS.getString(2); // The Returned priority ID
        ParameterList[1] = ResourceType; // Resource Type
        ParameterList[2] = Main_RS.getString(1); // The Returned System ID
        ParameterList[3] = Main_RS.getString(3); // The System IP/Queue Name

        SQLStmt= FormatSql("ResourceQuery",ParameterList);
    } // IF
    else
        System.out.println("**** Returned Result Set is Empty .....");

    ResultSet Sub_RS= dbServ.ExecuteQuery(SQLStmt,false);
    //System.out.println(SQLStmt);
    IntializeHashTable(RSTable);
    int RowCount = dbServ.RowCount;

    for (int i =0;i<RowCount;i++)
    {
        // Main Systems Handling
        RSTable = dbServ.GetQueryData(Sub_RS,RSTable);
        String ResourceCMDQuery = (String)RSTable.get("ResourceQuery");
        SysIP = (String)RSTable.get("IP");
        String QResult =null;

        if (ResourceCMDQuery.equalsIgnoreCase("null"))
            System.out.println("**** Problem : There is no Query for
"+(String)RSTable.get("ResourceQuery")+" system !!");
        else
        {
```

```

QResult = commandDispatcher.ExecuteCall(SysIP.trim(),ResourceCMDQuery);
if (QResult == null)
    System.out.println("**** Problem : Null Result for
"+(String)RSTable.get("OwnerSysID")+" system Query!!");

    // Update DB
    String [] ParameterList2 = new String[5];
    ParameterList2[0] = CurrentTime();
    ParameterList2[1] = QResult;
    ParameterList2[2] = (String)RSTable.get("OwnerSysID");
    ParameterList2[3]= (String)RSTable.get("ResourceID");
    String FinalSQL= FormatSql("UpdateResourceQuery",ParameterList2);
    //System.out.println(FinalSQL);
    dbServ.ExecuteQuery(FinalSQL,true);
} // ELSE
} // FOR LOOP

// Run the same query again but order by priority , resource usage
SQLStmt= FormatSql("ResAcqQuery",ParameterList);
ResultSet NegotiateRSet= dbServ.ExecuteQuery(SQLStmt,false);
RowCount = dbServ.RowCount;
// This is to loop on all the candidate systems to give up some resources
for (int i =0;i<RowCount;i++)
{
    // Start request from each system to give resources
    RSTable = dbServ.GetQueryData(NegotiateRSet,RSTable);
    String ResourceAcqCMD = (String)RSTable.get("ResourceAcqCMD");
    String QResult =null;
    if (ResourceAcqCMD.equalsIgnoreCase("null"))
        System.out.println("**** Problem : There is no CMD Query for
"+(String)RSTable.get("ResourceAcqCMD")+" system !!");
    else
    {

        if (Commit) // This to indicate execution mode and not testing mode
        {
            QResult = commandDispatcher.ExecuteCall(SysIP,ResourceAcqCMD);
            // A Zero QResult means the Transaction is successful
            if (Integer.parseInt(QResult) == 0)
            {
                String [] TmpParameterList = new String[5];
                TmpParameterList[0] =(String)RSTable.get("OwnerSysID");

```

```
    TmpParameterList[1] = ParameterList[2];
    TmpParameterList[2] = ResourceType;
    TmpParameterList[3]= CurrentTime();
    SQLStmnt= FormatSql("InsertTransQuery",TmpParameterList);
    dbServ.ExecuteQuery(SQLStmnt,true);
    //break;
    return(0);
} // QResult
return(99);
} // IF Commit
} // ELSE
} // FOR

} // try
catch (Exception e)
{
    System.out.println(e.getMessage());
}
return(-1);
} // Function

} // CLASS
```