September 2013

Collaborative Policy-Based Autonomic Management in IaaS Clouds

Omid Mola
The University of Western Ontario

Supervisor
Dr. Mike Bauer
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

COLLABORATIVE POLICY-BASED AUTONOMIC MANAGEMENT IN IAAS CLOUDS

(Thesis format: Monograph)

by

Omid Mola

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

With the increasing number of "machines" (either virtual or physical) in a computing environment, it is becoming harder to monitor and manage these resources. Relying on human administrators, even with tools, is expensive and the growing complexity makes management even harder. The alternative is to look for automated approaches that can monitor and manage computing resources in real time with no human intervention. One of the approaches to this problem is policy-based autonomic management. However, in large systems having one single autonomic manager to manage everything is almost impossible. Therefore, multiple autonomic managers will be needed and these will need to cooperate in the overall management. We propose a management model using multiple autonomic managers organized in a hierarchical fashion to monitor and manage the resources in a computing environment based on provided policies. We develop a communication protocol to facilitate collaboration between different autonomic managers, define the core operations of these managers and introduce algorithms to deal with their deployment and operation. We also introduce an approach for the inference of the communication messages from policies and develop several algorithms for joining and maintaining the management hierarchy. We propose a deployment system that can discover relevant resources in a computing environment automatically to facilitate the deployment of autonomic managers at different levels of a physical system. We then test our approach by implementing it in a small private Infrastructure-as-a-Service (IaaS) cloud and show how this collaboration of autonomic managers in a hierarchical way can help to adopt to high stress situations automatically and reduce the SLA

violation rate without adding any new resources to the environment.

Keywords: Cloud Computing, Autonomic Management, Policy-Based Management, Collaborative Management.

Dedication

This dissertation is lovingly dedicated to:

My lovely father, who encouraged me to continue my studies, helped me migrate to Canada and taught me to never give up.

My beautiful mother, who prayed for me every day, missed me but hid her tears and believed in me more than what I deserved.

The love of my life, Zeinab, who left her job so I can continue my studies, stood by my side through difficult times and supported me in this life journey.

My little son, Mahan, who should know that it is always possible to learn and gain knowledge, even when you have a baby at home!

Acknowledgements

A big special thanks to the best supervisor I have ever had. Without his patience, mentorship and support this work was not possible. I wish I could thank him enough for being my supervisor. Thank you Dr. Mike Bauer, now and always.

I would like to acknowledge and thank Dr. Hanan Lutfiyya, my supervisory committee member, who were always more than generous with her expertise and time and helped me with her constructive feedbacks to improve this work.

I wish to thank Dr. Mark Daley, my supervisory committee member, Dr. Miriam Capretz, Dr. Mike Katchabaw and Dr. Patrick Martin for being in the examining board. I appreciate your time, insights and valuable comments.

I am very grateful to the Computer Science department staff members, Ms. Cheryl McGrath, Ms. Janice Wiersma, Ms. Dianne McFadzean and Ms. Angie Muir who always helped me specially when I was teaching courses at Western. Thanks for making this experience wonderful.

I would like to thank all of my friends in our research group (DiGS) and Sharcnet administrators specially Mr. Nathaniel Sherry, who devoted their time to discuss and brainstorm different ideas related to this thesis. Your constructive criticisms contributed in improving this work.

Last but not least, thanks to all of my family members specially my lovely smart brother, Amir, who was always kind, passionate and keen about me. I love you all and feel blessed to have you around me.

Contents

List of Tables

List of Figures

List of Algorithms

List of Appendices

Chapter 1

Introduction

In recent years, there has been a lot of research into "Autonomic Computing" [17], especially about how to build autonomic elements and managers [19]. Autonomic managers (AMs) try to monitor and manage resources in real time to ensure that the components they manage are self-configuring, self-optimizing, self-healing and self-protecting (so called "self-*" properties [36]).

1.1 Motivation

The basic idea behind a self-management system is inspired from the autonomous nervous system of human body [39]. The need for having such systems is becoming more obvious as the number of computing machines (such as virtual or physical) is increasing. Data centers are becoming larger and more complex, particularly those focused on providing cloud services. The challenges of monitoring and managing these cloud computing environments in order to meet users' expectations of highly available and responsive systems are increasingly more difficult. Therefore, as the number of cloud users

are growing, having self-managed systems seems to be inevitable in the future management of the computing infrastructures.

In the broader vision of autonomic computing, large complex data centers and systems will consist of numerous autonomic managers handling systems, applications and collections of services [20]. Some of the systems and applications will come bundled with their own autonomic managers, designed to ensure the self-properties of particular components. Other managers will be part of the general management of the computing environment. Therefore, the complexity of managing a large system will entail a number of different autonomic managers which must cooperate in order to achieve the overall objectives set for the computing environment and its constituents. However, the relationships between these managers and how they cooperate introduce new challenges that need to be addressed.

More specifically, there are questions regarding how different autonomic managers should be organized and how they should interoperate in a large computing environment, such as an Infrastructure-as-a-Service (IaaS) cloud. There are questions on how they should interact with each other to achieve a global goal in the system, how and when this communication should happen, how to minimize associated overhead, etc. Besides communication among managers, there are other problems that need to be addressed: How each manager gets deployed in the appropriate position of the management system, how is the configuration and the deployment of the managers done so that they can collaborate with each other in the system, how is the management relationship among managers maintained as new

managers start and others end in response to changes in the system? In order to ensure that service level agreements are met and that we use the infrastructure more efficiently, we have focused on the following problems:

- How to deploy autonomic managers dynamically in a scalable manner?
- How autonomic managers should collaborate with each other in a large computing environment to achieve global goals?
- How to automate the configuration of autonomic managers and the communication process itself, to minimize the administrative costs of managers' setup and maintenance?
- What should happen when a new autonomic manager gets added or when an already running one stops working? How system react to these changes dynamically?

We consider the use of policy-based managers [6] in addressing these problems. The ultimate goal is to automatically monitor and manage a large system by a collective of collaborating local autonomic managers. In such an environment, we assume that each local autonomic manager has its own set of policies and is trying to optimize the behaviour of the elements that it manages by responding to the changes in the behaviour of those elements. We assume some managers will also be expected to monitor multiple systems and directly or indirectly to monitor other local AMs.

The focus of this research is on a management model for multiple autonomic managers and in particular the collaboration and communication between different managers. We consider this initially where

the autonomic managers are organized into a hierarchy and investigate how they can communicate at different levels of a hierarchy based on the active policies. Although there are other approaches to communications between managers, such as peer-to-peer, multi agent, etc. we have chosen a hierarchical approach since a) it is a good starting point and has advantages over flat structures and it is important to understand how it can be effectively utilized or where there may be limitations and b) it has a natural alignment with an IaaS cloud architecture - our particular system focus.

The core issues addressed are how these local managers should communicate with each other, how they should be deployed automatically across the computing environment and what information they have to exchange to achieve global performance goals. We will also focus on how to automate the collaboration process itself by inferring the communication messages from the active policies in a particular autonomic manager. In a hierarchical organization of autonomic managers, policies are used at different levels to help managers decide when and how to communicate with each other as well as using polices to provide operational requirements. We assume that one of the roles of a higher level manager is to aid other autonomic managers when their own actions are insufficient to meet operational requirements.

1.2 Towards Autonomic Cloud Management

A special focus of this thesis is on the management of IaaS cloud environments. These cloud computing environments often depend

on virtualization technology where client applications can run on separate operating virtual machines (VMs). Such environments can consist of many different host machines each of which might run multiple VMs. As the number of hosts, virtual machines and client applications grow, management of the environment becomes much more complicated. The cloud provider must worry about ensuring that client service level agreements (SLA) are met, must be concerned about minimizing the hosts involved, and minimizing power consumption.

As part of this thesis, we focus on how our management model and approach can be applied to such environments (e.g. IaaS clouds) and implement these ideas in a small cloud. We also explain the application of our approach to a real world problem where we worked with a private company to evaluate these ideas in a high frequency trading cloud environment and tuned the general strategy based on practical experiences.

1.3 Contributions

The main contributions of this work and the novel ideas are as follows:

- There has been generally a little work in the area of multiple autonomic managers and how to handle dynamic changes. Therefore, this work is to somewhat unique in this area.
- Cluster management typically has a focus on the cluster as a whole often ignoring management of individual elements, such

as nodes. Our hierarchical approach in this thesis encompasses a focus on local and intermediate managers as well as including global cluster level managers which makes it unique in addressing this problem.

- The design of a hierarchical autonomic management model for large computing environments with formal definition of different elements in that model (Chapter 4).
- The design of a communication protocol between autonomic managers that facilitates their collaboration in achieving global goals (Section 5.2). Some of these communication messages can be inferred from policies and therefore can help with automating the collaboration between managers.
- Introduction of multiple algorithms that define the behaviour of a specific autonomic manager and its relationship with other managers in that management model. These algorithms include the start-up, processing, termination detection and communication message inference from policies (Chapter 5).
- Design of a deployment system based on the management model proposed to automate the deployment of different autonomic managers across the computing environment with minimum administrative efforts (Chapter 6).
- Creation of multiple algorithms as part of this deployment system such as element discovery, members addition and members removal (Section 6.4 and Section 6.5). The time complexity of element discovery algorithm is $O(n^2)$ where n is the number of AMs that should be deployed in the whole computing environment (e.g. number of nodes in the management tree). The time

complexity of members addition algorithm is O(log(n)) and the members removal is O(n) in the worst case.

We also evaluated these ideas in two different experimental settings. In one case, we implemented this approach in a small private cloud and measured the potential advantages of a hierarchical approach. We also implemented some of our ideas and algorithms in a real world setting involving a high frequency trading cloud infrastructure.

1.4 Roadmap

The structure of the remainder of the thesis is as follows: Chapter 2 presents a literature review and a summary of the most related work. Chapter 3 explains some of the background information required and defines the scope of this research outlining the specific challenges that are addressed throughout the thesis. Chapter 4 explains the basic assumptions in our approach, discusses the proposed management model and introduces formal definitions of different elements in this model. Chapter 5 introduces and discuses the multiple algorithms that are developed to explain the behaviour of an autonomic manager and its relationship with other managers within the scope of the proposed management model. Chapter 6 addresses the issue of dynamic autonomic manager deployment in a large computing environment and explains several algorithms that are used inside the deployment system to make sure that managers are automatically deployed to the right position with the right configuration parameters. Chapter 7 describes the implementation and evaluation

of these ideas through a performance study and describes some practical experiences in using the ideas in this thesis in a private cloud environment. Finally, Chapter 8 provides a summary of the thesis and concludes by identifying some potential future work.

Chapter 2

Related work

There has been wide range of research dealing with the issues involving multiple autonomic managers for managing large systems. In this Chapter, we review some of the key works in this area and discuss the similarities and differences with our work. We focus in particular on previous work that looks at having multiple managers and examine how the interactions or collaboration are addressed. We also discuss some of the previous research that involves policy-based management with multiple autonomic managers, cloud management and general approaches towards coordination of multi-agent systems.

2.1 Multiple Managers

Some researchers have already begun to study how collaboration among local autonomic managers can be done in order to achieve a global goal. A hierarchical communication model for autonomic managers has also been used by some researchers. In this section, we describe some of the relevant research in this area and explain the differences with our work.

Famaey, et al. [14] used a policy-based hierarchical model for network management. They showed how this model can be mapped to the physical infrastructure of an organization and how this hierarchy can dynamically change by splitting and/or combining nodes to preserve scalability. They also introduced the notion of "context" that needs to be accessible in the hierarchy. The "context" is the information that is made available from a child to its father. Their work focused on the network layer and studied the transmissions between autonomic elements by looking at the number of bytes transmitted during communication. In this work, "context" is basically the monitoring information that can be retrieved from standard protocols such as the Simple Network Management Protocol (SNMP). The limitation of these protocols is that they only provide network management information at the macro levels and they do not deal with detailed organized information that is required for management at higher levels. It is also difficult to perform request/response type of communication due to protocol constraints. We have also adopted the hierarchical approach used in this work, but we develop a new protocol to exchange communication messages between autonomic managers. We also use a mechanism to infer communication messages from policies automatically and show when and how this communication should happen.

Aldinucci, et al. [4] described a hierarchy of managers dealing with a single concern (Quality of Service-QoS). In their work, each manager is trying to pursue a goal defined in a QoS contract. Therefore, the relationship between managers is bound to the contracts

they are pursuing. This means that policies are defined based on these QoS contracts and that the topmost contract is the main QoS for the whole system and other contracts within the hierarchy should be derived from their parents. This is similar to policy decomposition process where one can define global policies for the root node and then it gets decomposed to lower level policies and placed on to the lower level components. They assume that if a lower level component can not satisfy its QoS parameters, it will trigger a "contract violation" message to its parent and enter a passive mode until it receives a new contract. This means that the parent node should be able to generate a new policy set upon QoS violations to pass it down to its child. They used a simulator to evaluate the framework. The communication between elements in this work is hard-wired and the hierarchical structure is static which makes it difficult to deal with dynamic environments. The focus of our work is not on how the policies get distributed between different managers (though for completeness we describe an approach in our work), we assume that autonomic managers have the right policies in place and that these policies can change over time if required. We focus on defining a communication protocol between managers that is loosely-coupled and on what should happen in case of adding/removing a manager to the hierarchy which is not discussed in this paper. We consider dynamic join and leave of autonomic managers to the management hierarchy and design algorithms that can detect these changes and adopt the hierarchy accordingly.

2.2 Coordination of Managers

Mukherjee, et al. [32] used coordination of three managers working on three different parts of a system (Power Management, Job Management, Cooling Management) in a flat structure to prevent a data center from going to a critical state. The critical state is when there is a possibility of the ambient temperature to reach the redline temperatures. They showed how the three managers can cooperate with each other to keep the data center temperature within a certain limit that is suitable for serving the current workload and at the same time not using more power than required. They showed how these three managers can be configured to work based on different business policies. Their approach used three different strategies to combine the three management tiers and preconfigured the system to work based on these three strategies. The three management tiers are fixed and adding new managers to this system will be challenging both in terms of collaboration and scalability, particularly because the coordination between management tiers depends on their configuration and can not change dynamically.

The same approach as in [32] is used in [18, 49] to show the collaboration between a power and a performance manager (only two managers) to minimize the power usage as well as maximizing the performance. This method however does not seem to be generalizable to a larger environment with more autonomic managers involved because of the complexity introduced in terms of interactions between managers. In contrast, we look to deal with multiple managers and an environment where the managers can join and leave the management system dynamically. The configuration of managers is

not fixed and can be changed based on the active policies. The policies themselves can also change on the fly based on new demands rising from time to time.

Schmelz, et al. [44] have proposed a coordination framework for Self-Organizing Networks (SON). They use a coordinator (Alignment Function) to coordinate the decisions of multiple managers for a specific network entity based on predefined high-level performance objectives. This work is focused on resolving conflicting parameter settings for the network entities because SON functions (managers) are not necessarily aware of each other and may cause making multiple conflicting decisions for one specific element. This means that one element can be managed by more than one manager and therefore each manager might change the settings of the element without knowing about decisions of other managers. They used a policy decomposition framework to map and distribute high level performance objectives defined by network operators to cell-specific policies, SON function-specific policies and SON coordinator-specific policies. The SON coordinator will then resolve the conflicts based on these policies try to harmonise the control parameter changes towards the operator policies. In our work, we assumed that each managed element is being managed by one and only one autonomic manager and therefore the only possible way to have conflicting decision is when there are conflicting policies in place. Each manager might be involved in a relationship with its higher level manager and in case of conflicts in enforcing policies at different levels, we assume the higher-level manager has more authority and therefore its policy should override the local lower level manager's policy. However, the

policy distribution and how policies are derived from higher level objectives is not the focus of our work. If we assume the SON coordinator as a higher level manager then this system can be considered as part of the hierarchical system proposed in our work.

Tuncer, et al. [51] have developed a coordinated mechanism to control the distribution of traffic load in the IP networks. Their ultimate goal is to balance load in the network by moving traffic away from busy nodes towards underutilized ones in order to adapt to dynamic traffic changes. This work does not deal with autonomic managers, but they have explored two different models for the organization of nodes. They used full-mesh and ring topology models for organizing nodes in a decentralized way and developed an algorithm for their coordination. However, the nodes in these two models are fixed and they do not consider faulty/error situations. They also used a message based means of communication between nodes to facilitate the coordination. They created a structure for messages and defined two types of messages that can be exchanged (i.e. REQUEST, RESPONSE). This communication protocol can only handle simple messages with a focus on networking (i.e. it is not generalizable to another system). Our focus is on a hierarchical organization of autonomic managers with a message based communication. The experiences in this work such as defining different types of messages, and the communication protocol can be used in our work. However, we extended the communication protocol to include other types of messages such as NOTIFY message and also added useful organized information in the body of messages. We also deal with dynamic addition and removal of nodes in the system and pro-

vide algorithms to handle these cases.

Multi-agent approaches toward autonomic manager collaboration have been explored by some researchers. In these systems, each autonomic manager is represented as an agent and multi-agent communication techniques are used for their interactions. We explain some of the most relevant ones to our work.

The "Unity" architecture [50] uses performance utility functions that need to be calculated by each agent with the result being sent to a central coordinator ("Arbiter") for computing the globally optimal resource allocation. The same kind of approach is used in [12] by having a coordinating agent that tries to coordinate power and performance agents. This approach could be used as part of the hierarchical approach that will be presented in this thesis, but it does not seem to be scalable to a larger system just by itself, because adding a new agent to this system will introduce challenges in agent interactions and configurations. This can be considered as a special case of the hierarchical approach proposed later in the thesis, but with only one level of hierarchy.

Soares and Madeira [48] have used a multi-agent architecture for autonomic management of virtual networks. In this architecture each autonomic manager (agent) monitors part of the network and updates its own knowledge base (KB). In order to facilitate the decision making process, each agent should have access to the KB information of other agents so that it can get a global view of the system (network). Therefore, each agent should sync its own KB with all

other agents in the system. The agents do not request information on demand. This approach can lead to a major overhead in large systems and turn into a bottleneck itself. In our model, each manager does not need to have information about all other managers mostly because of the layered hierarchical architecture. It is also possible to get updated information on demand if it is needed.

2.3 Policy-Based Interactions

Salehi and Tahvildari [41] published a survey article on self-adaptive systems and the main research challenges. They proposed [40] a policy-based orchestration approach for resource allocation to different autonomic elements. They suggested the use of a global orchestrator to coordinate the resource provisioning at a global level between multiple autonomic elements. They used crisp action policies for non-competitive states where there is no conflict on resource requests and fuzzy utility policies to resolve conflicts in competitive states. In this work, the interactions between autonomic managers and the orchestrator is limited to resource requests, the managers are fixed and the communications are tightly coupled. This system can be considered as one level of hierarchy that will be explained in this thesis. However, we developed a new communications protocol which includes different types of messages and considered the dynamic joining and leaving of managers from the system.

Schaeffer-Filho, et al. [42, 43] have introduced the interaction between Self-Managed Cells (SMCs) that was used in building pervasive health care systems. They proposed "Role" based interactions

with a "Mission" that needs to be accomplished during an interaction. This mission is based on predefined customized interfaces for each role. This approach is similar to interactions in ad-hoc networks and SMCs need to discover each other and try to accomplish missions based on their defined role in the system. There are certain policies for each role which facilitate the interactions. In our work, we deal with a hierarchy of managers and therefore each manager needs to communicate with either its children or its parent and there is no need to define "role" for each manager to make the interactions possible. Although, this SMC role based approach might also be applicable in a hierarchical fashion, the overhead in defining unnecessary interfaces introduces a challenge. Another major difference is that we try to infer communication messages from policies dynamically to the extent that is possible but in the SMC system all interactions have to be specified beforehand through missions.

2.4 Cloud Management

Zhu, et al. [58] introduced an integrated approach for resource management in virtualized data centres. They used three controllers (e.g. Node, pod and pod set controllers) to monitor physical nodes, a cluster of nodes and the whole data center. Their approach is similar to the hierarchical approach we used in our work, but the relationships between different controllers are tightly coupled whereas we suggest a loosely coupled communication style to better accommodate failures, heterogeneous autonomic managers and dynamic changes in the system and managers. They also only focus on management of the physical elements; i.e., they do not consider any controllers that

could monitor the changes inside a virtual machine. In the application of our model to a cloud environment, it is possible to have at least one autonomic manager inside virtual machines which can join and leave the management hierarchy dynamically. Another difference is that they use a polling mechanism with multiple time-scales to do the monitoring at different levels. For example, they monitor virtual machines in seconds, the "pods" in minutes and the "pod sets" in hours or days. In our work, we proposed the use of notification messages for communication between managers and therefore managers can communicate based on demand rather than polling which reduces the overall overhead. We also focus on policies and how they affect the relationship between managers in a dynamic structure where multiple autonomic managers can join and leave the management system, but the number of controllers in their system is limited to three with hard-wired connections between them which limits the scalability of their approach.

Li, et al. [23] have developed an integrated and multi-layer approach towards automatic management of cloud environments. They used three different controllers to monitor and manage cloud infrastructure at three different layers. The virtual machine controller (VMC) uses a multi-input multi-output (MIMO) resource controller and a model estimator to estimate the required resource allocations of the applications running inside the VM to satisfy their service level objects (SLOs). The node controller (NC) collects all VM resource demands and satisfies these demands according to SLO differentiation which means lower priority applications will be given fewer resources if there are not enough resources available or when the

total demands are more than available resources at the node level. The global controller (GC) monitors all nodes in the cloud and uses a statistical machine learning technique to rearrange virtual machines among nodes. These migrations help to optimize virtual machine placements and meet the SLOs. Basically they use different models at each level to estimate the demands and adjust the environment and as a result it is more complex to change the model dynamically. In our work, we use the same Monitor-Analyse-Plan-Execute (MAPE) loop/model at all levels of the hierarchy but policies can change on the fly. In their work, the interactions between controllers are tightly coupled. For example, the VMC sends resource requests to NC and the NC responds to that request by using a resource actuator to change the VM parameters. This also shows the tightly-coupled relationship between controllers and can become an issue for scalability of the approach when the number of virtual machines and nodes increase. In our work, we develop algorithms to handle dynamic joining and departure of managers to the hierarchy and the message based communication protocol is loosely-coupled.

There has been some other research about management of the virtual machines in a cloud environment. Pokluda, et al. [38] looked into how to change VM's memory allocation dynamically in stress situations. Urgaonkar, et al. [53] developed an algorithm for dynamic resource allocations at the virtual machine level to adapt to unpredictable changes. Researchers in [55, 45, 57] have developed multiple algorithms for virtual machine migrations in data centers to address stress situations. However, in all of these works the approach was based on a single centralized manager that gathered all

required information from the virtual machines and made decisions based on that information. This approach is limited in terms of scalability and single point of failure. In our management model, we consider multiple autonomic managers deployed at different levels of the system. We consider dynamic joining and leaving of these managers so that the system can still operate if one or some of them are terminated. We also focus on a range of managed elements in the data center, such as applications, virtual machines, physical servers and so on but in these works the main focus is only on virtual machines. Overall, the algorithms developed in these works can be used as part of our management model and inside some of the managers that are responsible for managing virtual machines. These could be embedded in to the policy sets defined for those managers.

2.5 Summary

We explored some of the work most relevant to this thesis and discussed some of their limitations and constraints. We also explained how our work is different from each. In general, understanding of the collaboration between policy-based autonomic managers is still a relatively new area of research with a lot of research challenges yet to be answered. It is still a work in progress in the autonomic computing area; some areas where the previous work has not yet studied include:

1. A good communication protocol between managers that is loosely-coupled, can handle different types of interactions and can include detailed organized information as part of the communica-

tion.

2. An organizational model that can handle the dynamic joining and leaving of managers and adapt to the changes in the infrastructure.

3. Methods that address scalability concerns as the number of elements increase in the computing environment and automation of communication between managers to the extent that is possible.

4. Means of deploying multiple managers to the right position and keep them up to date and running with the least human administrative efforts.

We focus on these main issues in the rest of this thesis and explain our approach in more detail.

Chapter 3

Scope and Challenges

In this Chapter, we cover some of the background information for our work before describing our approach in more detail. This Chapter is meant to give readers a clear view of the scope and challenges addressed in this thesis.

We first explain the basic concepts of autonomic management and associated techniques used in this thesis. We then describe the IaaS (Infrastructure-as-a-Service) cloud architecture since it is the environment we focus on to explore our approach and test our ideas. This Chapter also provides some of the background behind why we have initially chosen to focus on the hierarchical structure for organization of autonomic managers. Moreover, the cloud architecture will be used in the evaluation of our approach and discussed further in the description of our experiments. We also discuss the main questions we have addressed in this thesis.

3.1 Autonomic Management

Autonomic Management has been a very active field of research in the past decade [20, 36, 17] and a variety of research challenges have been raised in this area [19]. It is used for service level guarantees [28], aggregated information monitoring [26, 25] and many other applications [27]. However, the main idea behind autonomic management is to build systems that are self-configuring, self-healing, self-optimizing and self-protecting and it is inspired from the human body's Autonomous Nervous System (ANS) [39].

The ANS gives our bodies the ability to adapt to dynamic changes in the environment around us automatically by sensing these changes, deciding what actions the body should take and enforcing those actions. Similarly, an autonomic manager which is responsible for monitoring and management of one or more elements of a computing system (i.e. Managed Elements - MEs) should be able to sense the changes in those elements, decide what actions need to be taken and enforce those actions, to adapt the whole system automatically.

The general architecture of an autonomic manager looks like Figure 3.1. In this architecture [33], the managed element provides some sensors and effectors/actuators to the manager. The autonomic manager can then monitor available metrics through these sensors and analyse the monitored information. It can then plan for a series of actions that needs to be executed, if any, and then execute those actions through the provided effectors. The manager will then keep monitoring those metrics to see the effects of its decisions in the previous management interval. This process is a feedback loop

called Monitor-Analyze-Plan-Execute (MAPE) loop.

There are various ways a manager can choose the best actions, but we use policy-based management in this thesis. Policy-based management assumes that the knowledge base in the autonomic manager includes defined policies and therefore it can look into those provided policies to pick the appropriate actions that need to be enforced. Policy-based management is explained in more detail in section 3.1.1. The autonomic manager combined with one or more managed elements is called Autonomic Element (AE). Therefore, in order to be able to manage an AE itself, the AE can provide sensors and effectors to the outside world. This will help forming multi layers of autonomic management in a system.

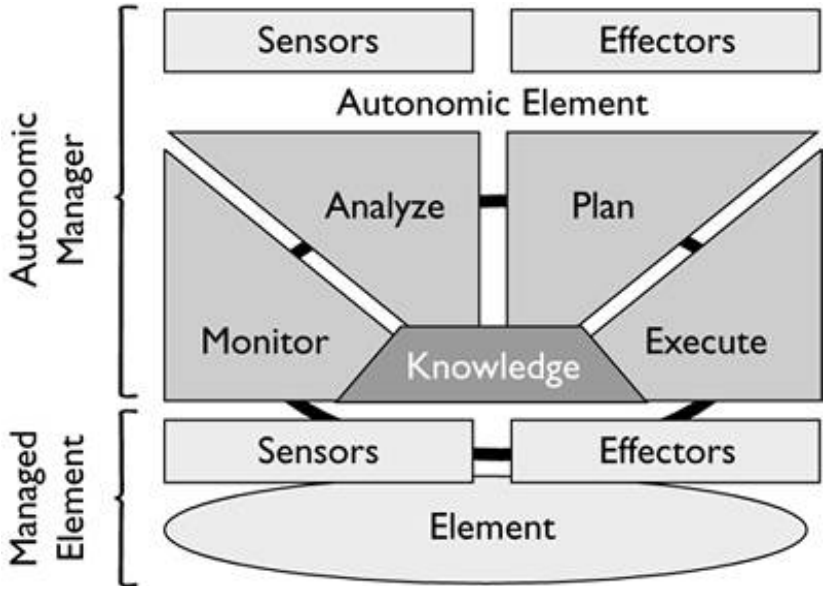

Figure 3.1: Autonomic Manager Architecture (from [33])

In this work, we use the same basic architecture for building autonomic managers and the MAPE loop can be configured to run on

different management intervals. That is, one manager can run its MAPE loop every 100 millisecond whereas another manager can run the loop every 10 minutes. This is useful for enforcing management at different levels of the hierarchy while trying to minimize overhead. At the lowest levels, monitoring is required more often as the changes are very dynamic and happen more frequently, while managers at higher levels need less frequent monitoring and therefore can in principle operate at higher management intervals. This will result in less traffic and processing overhead in the management system.

3.1.1 Policy-Based Management

Policy-based management is a well-known technique in the autonomic management area. An overview of policy-based management along with relevant standards and implementation techniques can be found in [3, 6]. Many languages have been developed to express policies however only some of them support Event-Condition-Action paradigm [16]. Ponder [11] is one of the most famous policy languages that supports this paradigm.

An autonomic manager can have different types of policies which can be useful for certain purposes. For example, it might rely on configuration policies for self-configuration of managed elements, or might utilize expectation policies for optimization of the system or for ensuring that service level agreements (SLAs) are met. Elasticity policies [15] can also be used to automatically add or remove resources in a computing environment.

In this work, we use policies expressed as event, condition, action (ECA) policies. In general, all of our policies are of the form:

OnEvent: E
if Set of Conditions **then**
    Set of ordered actions
end if

Upon raising an event inside the autonomic manager, then any policy which matches the event will get evaluated. If the conditions in the policy are met, then the policy actions get triggered. We provide examples of policies in the following sections.

At AM start-up there are configuration policies that set up the autonomic manager environment, identify the appropriate managed elements and configure them. It is however possible to automatically perform policy mapping [35] at start-up time and configure elements based on other types of policies. A sample configuration policy explicitly defined in the policy set of an autonomic manager would look like:

OnEvent: VMConfigurationEvent
if true **then**
    VirtualMachineMEI.setRefreshInterval(4000ms)
end if

This policy happens on AM start-up and configures the refresh interval for this AM. If we assume that this AM is responsible for

checking the CPU utilization, then the intent of this policy is that the AM will check the utilization every 4000 milliseconds and execute policies after each refresh.

These policies represent how the management system should react to dynamic changes in the environment and might change from time to time based on a strategy-tree approach [46, 47] or administrative needs. However, the focus of our work is not on how the policies get distributed between different managers (though for completeness we describe an approach in our work), we assume that multiple autonomic managers can retrieve their policy sets from a repository and that these policies can change over time if required.

3.2 Cloud Architecture

The focus of this thesis is on autonomic management and communication among multiple autonomic managers. However, while the focus is on autonomic management, we want to explore our ideas in an environment that has some structure. Given the importance of cloud computing environments, we choose to focus on this environment and, more specifically, we choose IaaS clouds to test our ideas.

The infrastructure of IaaS cloud providers is typically composed of data centers with hundreds to thousands of physical machines organized in multiple groups or clusters. Each physical machine runs several virtual machines and the resources of that server are shared among the hosted virtual machines. There are a large number of virtual machines that are executing the applications and services of

different customers with different service level requirements (via Service Level Agreement (SLA) parameters).

To have a better understanding of a cloud provider environment and architecture, we take a closer look at Eucalyptus [34] (an open source software for building private and hybrid clouds). There are three main distinct components that form the Eucalyptus architecture in a hierarchical fashion and each of the components have a different role in the system. These separate components can be physically located on one single machine to form a cloud or can be distributed over several machines. An overview of the Eucalyptus architectural model is illustrated in Figure 3.2.

The main three components of the Eucalyptus architecture are briefly described below:

- Cloud Controller (CLC): The CLC is the top level component for interacting with users and getting the requests. It handles all incoming requests and performs high level resource scheduling and system accounting. The CLC makes the top level choices for allocating new instances of virtual machines, authentication, reporting and quote management. Only one CLC can exist in each cloud.

- Cluster Controller (CC): The CC manages the virtual machine execution and service level agreements. It decides which node should run the VM instance. This decision is based upon status reports which the Cluster Controller receives from each of the nodes. CC has three primary functions: schedule incoming instance run requests to specific nodes, control the instance

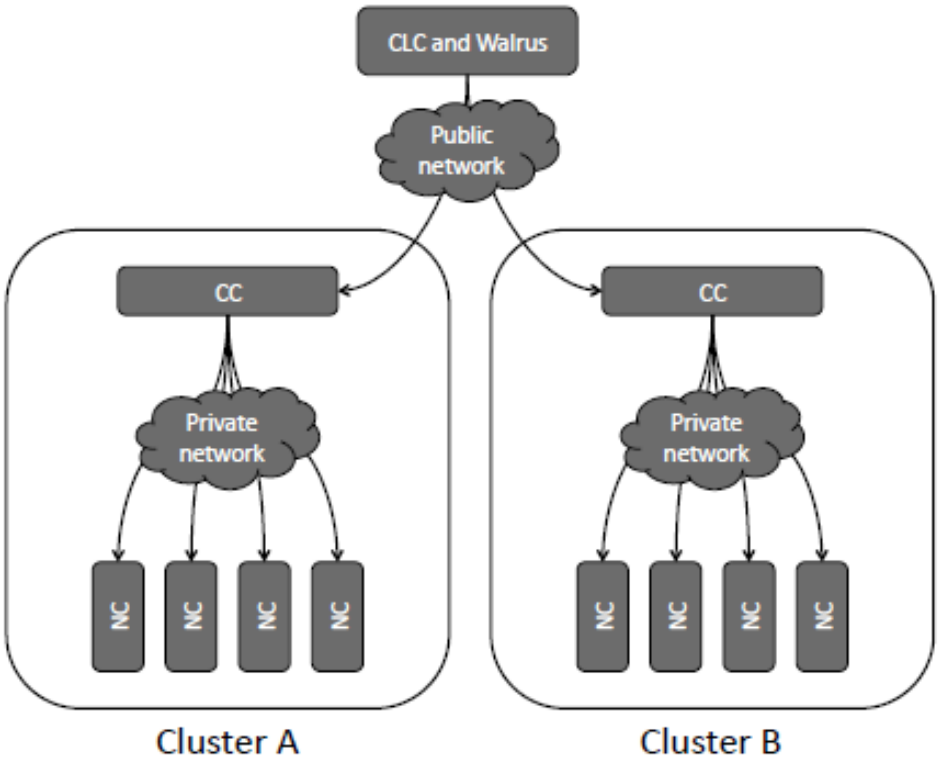

Figure 3.2: Eucalyptus Hierarchical Architecture (from [34])

virtual network overlay, and gather/report information about a set of nodes.

- Node Controller (NC): The NC runs on the physical machine responsible for running virtual machines and the main role of the NC is to interact with the OS and hypervisor running on the node to start, stop, deploy and destroy the VM instances. An NC makes queries to discover the node's physical resources the number of cores, the size of memory, the available disk space as well as to learn about the state of VM instances on the node. The information thus collected is propagated up to the Cluster Controller in response to "describeResource" and "describeInstances" requests.

This architecture shows a hierarchical relationship between different components of a typical IaaS cloud. A deeper look at the cloud architecture and the management needs suggest that providing management capabilities in real time through a single centralized manager is almost impossible, because of the hierarchical layers in the architecture with different responsibilities at each layer. Also, the dynamics of load changes and the need to react to these changes in real time with increasing number of virtual machines and physical nodes makes it much more difficult to achieve these goals with a traditional centralized manager.

We adopt the same hierarchical approach towards the autonomic management of this infrastructure. We organize policy-based autonomic managers in a hierarchical fashion which corresponds roughly to the underlying infrastructure components. Hence, the overall management of the system is then possible by having a set of collaborating autonomic managers organized in this hierarchy. At the same time, each manager in the hierarchy acts autonomously to manage part of the cloud on its own, based on given policies.

3.3 Challenges

All of the specified components in the cloud architecture are needed for instantiation of new images or destroying currently deployed virtual machines and they have some minimal management capabilities. The main job of these components is to pick the best host to place a new VM (VM placement). There has been a lot of research about VM placement [8, 37] which is usually relevant to the

problem of server/VM consolidation [9, 22, 54]. However, the main challenges in monitoring and managing the cloud environment occur after the virtual machines are placed and start working and receiving loads. After a virtual machine is placed with some specific service level agreements and starts working, the clients can connect to it for servicing. The number of clients and their interactions with the applications on the virtual machine will vary and create a dynamically changing workload. There may be times that the traffic is too high and the virtual machine gets overloaded or there may be some other times that the traffic is too low so that the virtual machine is underutilized. In the first case, SLA violations might occur whereas in the second case the energy and allocated resources might be wasted. One of the approaches to this problem is dynamic consolidation of VMs which usually is a heuristic based approach and does not allow explicit specification of QoS metrics [5].

In the context of an IaaS cloud, these problems are compounded with multiple virtual machines, multiple different applications and different service requirements. There are also many related management challenges that need to be addressed [56], including, how to initially configure and deploy autonomic managers, how multiple managers located at different parts of the system should communicate, etc. In this thesis, we focus on the following problems:

- How should multiple autonomic managers collaborate in order to manage the varying workloads in different virtual machines? To answer this question we need to understand what should happen to maximize the performance of a specific virtual ma-

chine (or an application inside it) according to the agreed SLA while minimizing the resource usage. We should also know if there is any way to get help from another manager while one manager has reached its local limits (e.g. communication and collaboration).

- How can we achieve autonomic elasticity in the cloud? Autonomic elasticity happens when a virtual machine can grow and use more resources if needed, and shrink back again and release resources if there is no demand for them. The answer to this problem would show what autonomic managers are needed, where they should be deployed and what kind of policies are required on each one.

- How to automate the collaboration of managers in the system? In order to deal with a dynamic environment where applications can start and stop and where virtual machines may come an go, there is a need to ensure that managers can communicate and collaborate. However, the interaction between managers must be dynamic too. How can communication between managers be defined in a changing environment as managers come and go? How is the communication structured and what information is exchanged (e.g. communication protocol). We look at a means of inferring the communication messages needed between different autonomic managers from their active policies.

- What is a scalable approach for the deployment of autonomic managers? In an IaaS system, there will be many autonomic managers that need to be deployed on different parts of the cloud, each monitoring some number of managed elements, and

the managers will change dynamically as applications and virtual machines come and go. What is a good strategy for deploying these managers so that it requires minimal manual administrative efforts?

- How can autonomic managers detect the addition or removal of different elements and automatically restructure their organization (e.g. hierarchy) without human intervention? In a real cloud, applications, virtual machines and physical nodes can join or leave the system at any time and thus their related autonomic managers can also join or leave the management system at any time. So, how does the organization of managers (e.g. hierarchy) restructure on the fly to reflect these changes?

- How to automate the manager configuration and minimize the administrative costs to setup autonomic managers? Each autonomic manager needs to be configured before or upon start-up. However, in a large system configuring all managers one by one can become a challenging and error prone job for administrators. How can this process be automated to help administrators and reduce the costs associated with it?

These problems can basically be categorized into two main areas: 1) Dealing with multiple managers: What managers are needed? How they are organized and communicate? What policies are required? When they should communicate? etc. 2) Management of the managers: How to address concerns arising from the management of the mangers: configuration, deployment, dynamic changes to the collection of managed elements, etc.

Chapter 4

Approach and Model

In this Chapter, we explain our approach and assumptions towards autonomic management of a large system (e.g. IaaS cloud) with a particular focus on the challenges outlined in Section 3.3. We propose a hierarchical model and provide definitions of different elements in this model.

Based on the previous discussions, we propose to use a number of different autonomic managers at different parts of the system. By doing this, the problem of managing a large system entails a number of autonomic managers where each one is dealing with smaller or more localized elements. Then each manager's job is to focus on managing that element (or small set of elements) efficiently based on certain policies. For example, an autonomic manager for an Apache web server should only focus on the behaviour of the web server itself and not the performance of the machine that this server is running on or, the autonomic manager for a Node Controller (NC) should only focus on the performance and behaviour of that specific node.

4.1 Assumptions

In this Section we describe the general assumptions we have for a management system that consists of multiple autonomic managers working collaboratively to achieve certain goals.

We assume that inside each autonomic manager there is an event handling mechanism for processing and generating events and notifying the interested parties inside that manager. For example, there could be an event bus and different components within the autonomic manager (AM) subscribe to certain events and upon raising any of those events, the subscriber would get notified. This event handling mechanism is useful for handling event-condition-action policies and also for communication between managers.

We assume each autonomic manager operates based on a set of policies provided to it. These policies could be a decomposition of global business policies [13] down to operational policies as suggested in [7, 10] or they could be given to different autonomic managers manually. We assume that policies can change over time, if needed. The focus of this research is not on exploring different approaches for how policies are distributed to managers. Rather, we assume one approach for our work, namely, that all policies are stored in a central repository and can be retrieved by an AM upon request. Each AM can retrieve its own set of policies on start-up and get updates each time policies change in the repository. We assume that each AM can evaluate these policies with a policy engine, which is basically a rule engine that provides the tools to evaluate rules (e.g. policies) based on available facts (e.g. latest values). Algorithm 5.4

shows the usage of this policy engine.

We assume that there is a central registry and that each AM will contact this registry during its start-up process. This registry will be used by each manager to find the contact information (e.g. ID in Definition 6) of its parent in the management hierarchy and is used to facilitate the process of adding new managers to the system dynamically.

In order to remove the single point of failure for registry and policy repository and also to increase the availability of the system, one can have backups (e.g. registry, repository) running at the same time which can be replaced upon a failure. This is beyond the current focus of this thesis.

We also assume that each manager should provide an interface for receiving messages from other managers. This interface should be able to receive different message types, parse them and do the proper actions according to the specification of that message. The message format and types are explained in more detail in Section 5.2.

4.2 Hierarchical Model

There are many ways to organize multiple autonomic managers in a large computing system (e.g. peer-to-peer, ring topology, etc.), but we focus on and explore a hierarchical management system to organize autonomic managers which might appear as in Figure 4.1.

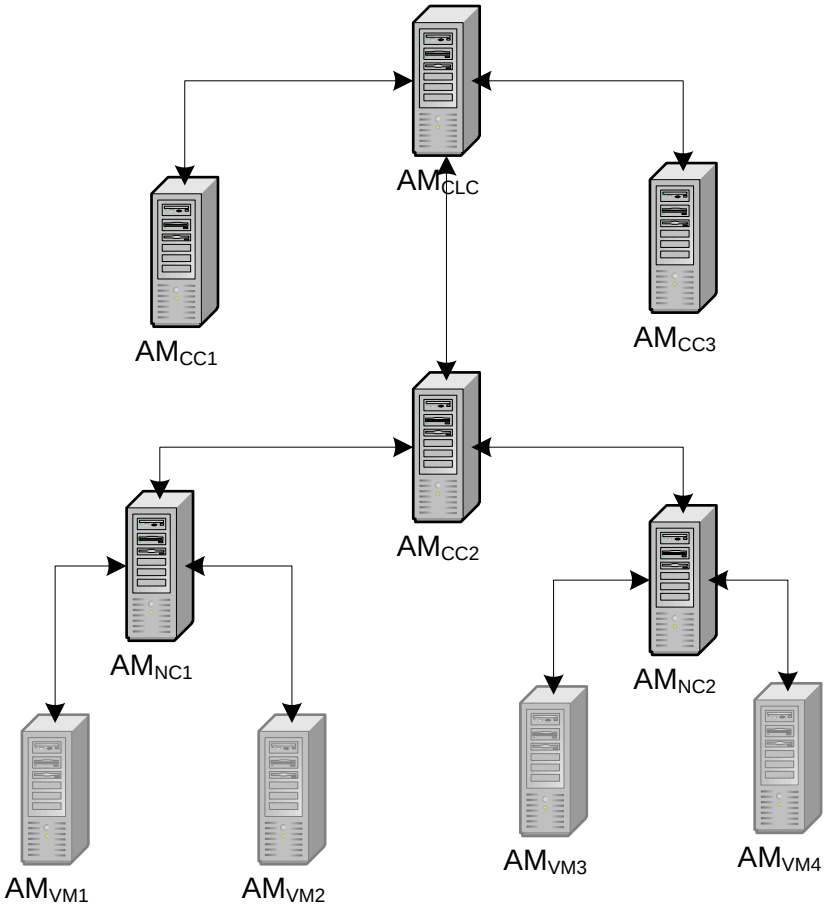

Figure 4.1: AMs hierarchy based on the cloud architecture

We choose the hierarchical approach because it is straightforward and a good starting point to explore collaboration between autonomic managers. A hierarchy provides a simple, yet useful, structure and has several advantages over a flat structure (e.g. improved scalability by reducing communication overhead that only happens between parent and child). This hierarchical model is also in natural alignment with the architecture of a typical Infrastructure-as-a-Service (IaaS) cloud which is our particular system focus.

The physical structure of a typical IaaS cloud would look like Figure 4.2. In this layout, every host machine is shared among multiple virtual machines and there could be many applications running inside each virtual machine. The host machines are grouped together to form a cluster and a combination of these clusters will form the cloud.

The management hierarchy can be expanded into more levels if needed. It can represent either the physical structure, logical structure or a combination of both in the computing environment. At the lowest level of our example management hierarchy in Figure 4.1, the autonomic managers are managing virtual machines and applications running inside them. It is, however, possible to have other application specific autonomic managers which will be located under these managers in the hierarchy, but for our initial work in exploring communication among automatic managers in an IaaS cloud, we have opted to consider autonomic managers of the virtual machines as the lowest level.

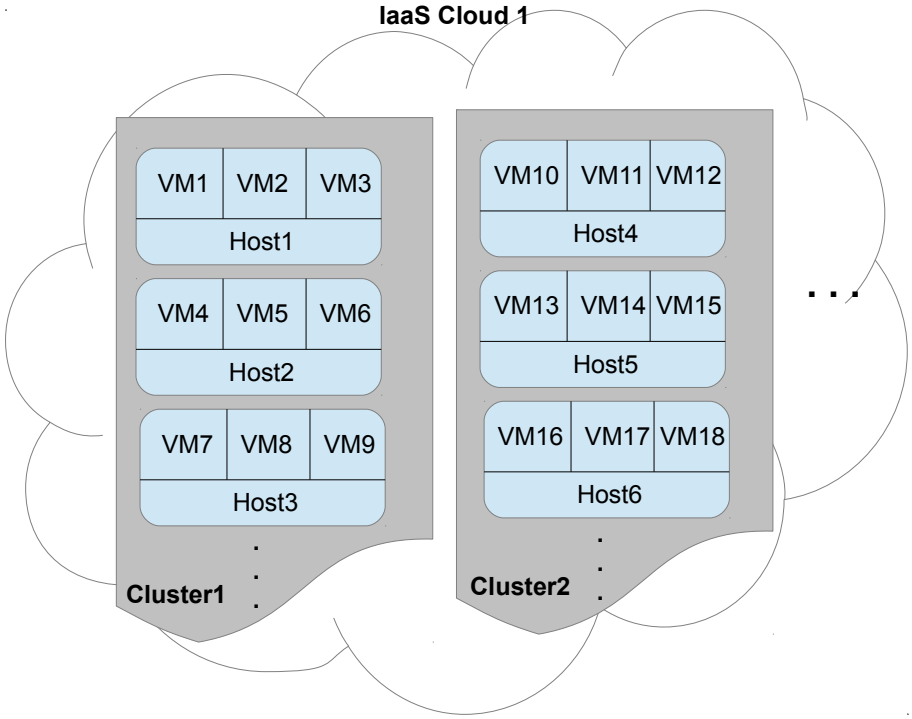

Figure 4.2: IaaS Cloud Structure

The AMs at the node controller (NC) level monitor and manage the physical nodes. An AM at the NC level can monitor the overall performance of the physical node and adapt to stress situations as much as possible. An AM at this level might have interactions with the AMs running at the virtual machine level to get updated monitored information or to request changes to happen inside virtual machines. This AM is aware of virtual machines specifics that are hosted by this node and it can allocate more resources (e.g. memory, cpu cores) to the stressed virtual machines based on availability of local resources or service level differentiations (e.g. Gold virtual machines get more resources than Silver ones in case of stress).

Then the AMs at the cluster controller (CC) level are responsible for monitoring a cluster with all physical nodes inside it. These AMs have a global view of the whole cluster and know which nodes are overloaded with traffic and which nodes are underutilized. In case of a virtual machine migration, these AMs can decide where should be the destination of the candidate VM for migration and inform the appropriate child AM to perform the migration to the selected destination.

Similarly, the AM at cloud controller (CLC) level monitors and manages all of the clusters. The overall monitoring of the whole cloud can happen at this level. This AM is the main entry point for defining business policies for the cloud. In case one cluster is overloaded and there is an underutilized cluster, this AM can choose that cluster and ask the overloaded child (Autonomic Manager) to offset some of the load to the underutilized cluster by migrating a few virtual machines to that cluster.

This is a logical organization of autonomic managers and does not necessarily reflect the physical allocation of the AMs, i.e., they do not necessarily need to be located on different physical machines. In a large cloud they could be located on separate machines or some may be located on the same machines. These AMs should then collectively work together to ensure that policies are met, e.g. policies for optimizing performance, minimizing resource usage, avoiding SLA violations, etc.

For management to happen in this hierarchical model, the big or

more complex tasks should be divided into smaller tasks and delivered to different responsible managers at lower levels. For example, the AM at the Cloud Controller (CLC) level should take care of balancing the load between different clusters and the AM at the Cluster Controller (CC) level should look after balancing the load between different nodes inside that cluster. Similarly, the AM at the node level should optimize the resource usage of that physical machine among different VMs and while the autonomic manager inside a VM should work on optimizing the applications performance. Assuming that the management "tasks" are specified in terms of policies, this means that we need policies with different granularity deployed at different levels of the infrastructure and we need to ensure that AMs can communicate properly with each other to enforce those policies.

These autonomic managers can be added or removed from this hierarchy based on demands of the computing infrastructure. Therefore, any particular hierarchy of AMs is not fixed and can change over time depending on what managers get created or removed dynamically. Automatic deployment and removal of AMs is a very important feature in order to minimize the impact of the management system on overall system performance and so that administrators do not have to worry about the hierarchy configuration every time there is a change in the infrastructure. The management system should be able to adapt to the infrastructure changes and automatically reconfigure itself as changes happen. We will explain this process in more detail in Chapters 5 and 6.

The policies for AMs can be defined via a repository, as we have

assumed, by the administrators directly or by some other process. A good "rule of thumb", however, would be to define similar sets of policies for autonomic managers that manage the same kinds of entities, e.g. the AMs that manage virtual machines would have similar policies, those that manage physical nodes would have similar policies. The rationale for this is that managers of the same kind of entity will have many similar or identical policies, e.g. a set of policies for managers of virtual machines might make use of the same policy to handle the situation when a VM has insufficient computing resources to meet an SLA. This way, sets of policies can then be stored in the policy repository and be retrieved upon AM start-up based on the kind of entity that the AM is managing.

4.3 Defining Elements of the Model

In this Section, we define various elements of our model. These definitions help to make concepts clear and we also use them in our algorithms and operations introduced in subsequent chapters.

4.3.1 Managed System

Our managed system is composed of a set of elements that can be monitored and managed automatically. Each autonomic manager is typically monitoring and managing one or more managed elements (ME). The managed elements could be equivalent to what is found in ordinary cloud infrastructures such as a virtual machine, a physical node, a software resource, or a cluster.

We assume the supported characteristics and operations of each ME is defined in a *ManagedElementInfo*, which is used to define the policies. For example, the AM responsible for managing a virtual machine, could be provided information in a *VirtualMachineManagedElementInfo*, which would include all supported metrics and actions of a general virtual machine. This is like a class definition for a specific ME which is used by the manager for policy definition. An instantiation of this class can then be used to evaluate the policies.

Definition 1 *A ManagedElementInfo, MEI, is a tuple* $MEI=\langle M, A\rangle$, *where:*

- *M is a finite set of metrics,* $M=\{M_1, ..., M_l\}$, *where:* $\forall M_i \in M, M_i = \langle N_i, T_i\rangle \mid N_i = Identifier(MetricName), T_i = MetricType \in \{String, int, double, ...\}$
- *A is a finite set of possible actions,* $A=\{A_1, ..., A_m\}$.

We denote the finite set of MEI by $MEISet=\{MEI_1, ..., MEI_n\}$

Actions are supported operations that can be done on that managed element. For example, actions for a *VirtualMachineMEI* could be *Shutdown*, *StartService*, etc. The metrics associated with a *MEI* include both attributes like *VirtualMachineName* which is a string, and metrics such as *CPUUtilization* and *MemoryUtilization* which are floating point numbers (e.g. type double).

Moreover, an *MEI* can be "subclassed" from another *MEI* to preserve reusability. For example, a *KVMVirtualMachineMEI* and

XENVirtualMachineMEI can both inherit actions and metrics/properties of a *VirtualMachineMEI* like virtual machine name, CPU utilization, etc.

Once an *MEI* is defined, it can be instantiated several times based on the need - an instantiation is referred to as a *ManagedElementObject*, defined below. For example, after a change in *CPUUtilization* or upon receipt of an event a *VirtualMachineMEI* can get instantiated with the latest facts/values and be passed to the policies for evaluation. This is done as part of the "Monitor" phase in MAPE loop (see Section 3.1). Basically, in order to monitor (gather information through sensors) a specific managed element the AM can instantiate its *MEI* and update the metrics that are available for that element.

Definition 2 *Given a set MEISet, a ManagedElementObject (MEO) is a tuple* $\langle m, a \rangle$ *where there is a MEI=*$\langle M, A \rangle \in$ *MEISet such that*

- a=A,
- $m = \{\langle N_1, V_1, T_1 \rangle, ..., \langle N_l, V_l, T_l \rangle\} \mid M = \{\langle N_1, T_1 \rangle, ..., \langle N_l, T_l \rangle\}$ *and* V_i *is the value of a metric.*

We denote the set of managed objects by MEOSet=$\{MEO_1, ..., MEO_n, ...\}$

A *MEO* is an instance of a *MEI* and represents actual values of a managed element information in the system. The metrics in an *MEO* are those in the definition of the class and the V_i are values associated with those metrics. Each V_i would typically be a value obtained by measuring some aspect of the actual managed element.

These MEO_i are used for policy evaluation from time to time.

The metrics and actions defined inside a *ManagedElementInfo* can be used in defining policies. For example, a *VirtualMachineMEI* can have a "CPUUtilization" metric and a "StopService(serviceName)" action both defined in its *MEI*. *CPUUtilization* represents the CPU utilization of the virtual machine which gets updated from time to time and *StopService* action takes a service name and stop that service from running. Therefore, an example policy to manage virtual machine stress can be:

OnEvent: ManagementInterval
if VirtualMachineMEI.CPUUtilization > 85 **then**
    VirtualMachineMEI.StopService("XY");
end if

This policy get executed at each management interval (e.g. upon raising *ManagementInterval* event), and checks the *CPUUtilization* of the virtual machine and if it is above 85%, it stops a service called "XY". Other possible *ManagedElementInfos* are: *ApacheMEI*, *HostMachineMEI*, *ClusterMEI*, etc.

Therefore at the time of policy definition, administrators use an *MEI* (e.g. one can think of it as a "class") to define policies but at run time the instantiated *MEI* (e.g. *MEO*) is passed to the policy engine to evaluate the policy's condition and perform actions. It is the job of the policy engine to match the *MEO* values with the right *MEI* metrics defined in the policies. The policy engine is the rule engine that evaluates the policies based on provided *MEOs*. This

happens inside the manager and Algorithm 5.4 described in Chapter 5 defines this policy evaluation process.

4.3.2 Events

We assume that inside each autonomic manager there is an event handling mechanism for generating events and notifying the interested parties (such as policy evaluator) inside the AM. For example, there could be an event bus and different subscribers to certain events and upon raising those events any subscribers will get notified. This event handling mechanism is useful for handling event, condition, action policies and also for communication between managers. We assume that for a given system and *MEIs*, that there are a finite number of event types.

Definition 3 *Given a set MEISet, an event type, Et, is a pair* $\langle N, M \rangle$ *where:*

- *N is the name of the event type,*
- M=$\{m_1, ..., m_o\}$, *and* m_i *is the name of a metric from an MEI* $\in$ *MEISet.*

We denote the finite set of event types by ET =$\{Et_1, ..., Et_z\}$.

Definition 4 *Given a set ET, an event E is a pair* $\langle n, m \rangle$ *where there is an event type* $Et = \langle N, M \rangle \in ET$ *and*

- n *is the name of the event* $n = N$,
- m =$\{\langle m_1, v_1 \rangle, ..., \langle m_o, v_o \rangle\}$, *where* M =$\{m_1, ..., m_o\}$, *and* v_i *is the value of* m_i.

We denote the set of events by EventsSet $=\{E_1, ..., E_p, ...\}$.

For a given set of event types, there may be an infinite number of possible events, depending on the values associated with the metrics of that event type. In this respect, an event is an instantiation of an event type with the associated metrics assigned values.

One sample event is *ManagementInterval* event, which is a simple event with no metrics that gets triggered on a time interval to trigger management loop.

E1 $= \langle ManagementInterval, null \rangle$

Another sample event is *HelpRequest* event, which can have one or more metrics attached to it. In this example, *CPUUtilization* of a virtual machine is attached to this event.

E2$=\langle HelpRequest, \{\langle VirtualMachineMEI.CPUUtilization, 95\rangle\}\rangle$

4.3.3 Policies

All of the policies are expressed as event, condition, action (ECA) policies. In general, all of our policies are of the form:

PolicyName: N
OnEvent: E
if Set of Conditions **then**
    Ordered Set of Actions
end if

Upon raising an event inside the autonomic manager, then any policy which matches the event will get evaluated. If the conditions in the policy are met, then the policy actions get triggered. We provide examples of policies in the following sections.

Definition 5 *Given a set MEISet and a set of events types ET, then a policy is a tuple* $\langle N, E, C, A\rangle$ *where N is the policy name,* $E \in ET$ *is one of the event types, C is a finite conjunction of conditions, and A is an ordered set of actions defined in* $MEI \in MEISet$*. Therefore,* $Pl{=}\langle N, E, C, A\rangle$*, where:*

- $E \in ET$,
- $C{=}\{C_1, ...C_p\}$ *and* $C_i{=}\langle MName, Operator, T\rangle$ *or "true", where MName is the metric name, Operator is a relational operator and T is a constant indicating a threshold value,*
- $A{=}\{A_1, ..., A_q\}$, $\forall A_i \in A$, $\exists\ MEI_j \in MEISet \mid A_i = MEI_j.A_k$

We denote the set of policies by $PL{=}\{Pl_1, ..., Pl_r\}$.

PL is the set of all available policies in the management system, but each autonomic manager would have its own subset of policies. Upon raising an event inside the AM, it checks all of its own policies and if a policy event E matches the raised event type then policy conditions will get evaluated based on the latest monitored metrics available in the relevant *MEO* and if they satisfy to true then it will take the policy actions based on the order in which they are defined. A single "true" condition implies that the actions should always be taken. A sample expectation policy for monitoring the Apache response time is:

$Pl_1 = \{$ *"ApacheResponseTimePolicy",*
ManagementInterval,
$\{ApacheMEI.ResponseTime > 500\}$,
$\{ApacheMEI.IncreaseMaxClients(25)\}$ $\}$

In this policy, *ManagementInterval* is an event that gets triggered in a certain time interval (e.g. every 1500ms) and it has no metrics associated with it. *ApacheMEI* is the managed element information for Apache and *ResponseTime* is one of the metrics defined in it. *IncreaseMaxClients* is one of the actions defined in *ApacheMEI* and will increase the *MaxClient* property of the Apache web server by a certain number (in this case 25).

At AM start-up there are configuration policies that set up the AM environment. A sample configuration policy would look like:

$Pl_2 = \{$ *"StartUpConfPolicy",*
Configuration,
$\{true\}$,
$\{VirtualMachineMEI.RefreshInterval{=}5000\}$ $\}$

This policy happens on autonomic manager start-up (once the *Configuration* event is raised) and configures the refresh interval of the AM. The AM will then refresh available metrics every 5000 milliseconds.

4.3.4 Structural Relationship of Autonomic Managers

In order to explain the relationship between autonomic managers, we first need to define the AM itself.

Definition 6 *Given a set MEISubSet* $\subset$ *MEISet, a set of event type ETSubSet* $\subset$ *ET and a set of policies PLSubSet* $\subset$ *PL, an Autonomic Manager(AM) is a tuple* $\langle ID, Name, MEISubSet, ETSubSet, PLSubSet, MI, RI\rangle$ *where ID is the AM's unique identifier which other AMs can use for communication purposes, Name is the AM name, MI is the management interval which determines the time interval for triggering the management loop and RI is the refresh interval which determines the time interval to refresh metrics of the managed elements in MEISubSet. We denote the set of AMs by AMSet=*$\{AM_1, ..., AM_t\}$

The autonomic manager *ID* is a globally unique identifier among all AMs and can be changed from time to time. This can be a URL or a physical IP and port where the AM can be accessed. The autonomic manager *Name* is a string and is set as a configuration parameter. This name is mapped to the AM ID and will be stored in the registry to be used to dynamically discover AMs for connection purposes. As part of the start-up process (Algorithm 5.1) each AM should register its name and ID in the registry.

Since AMs are organized in a hierarchical manner to reflect different authority levels, the structural relationship between them consists of a tree.

Definition 7 *Given an AMSet, a Hierarchy H of AMs is a tuple* $\langle AMSet, Edges \rangle$ *where AMSet is the set of autonomic managers as the nodes of the tree and Edges=*$\{(AM_i, AM_j) | AM_i, AM_j \in AMSet\}$ *is the set of edges connecting two AMs to each other. The following properties exist in this hierarchy:*

- $\exists\ AM \in AMS \mid \nexists\ AM_i \in AMS,\ (AM_i,\ AM)$
- *if* $(AM_i,\ AM_j) \in Edges \Rightarrow \nexists\ AM_k \mid (AM_k, AM_j) \in Edges$
- *if* $(AM_i,\ AM_j) \in Edges \Rightarrow (AM_j, AM_i) \notin Edges$

This definition means that there is at least one root manager in the hierarchy and for each AM there is only one parent. Also if a manager is the parent of another AM, then it cannot be that AM's child (e.g. there are no loops).

4.4 Summary

We have introduced a hierarchical management model and explained the assumptions for this system. We also defined different elements that are part of this model. In order to address challenges explained in Section 3.3, each autonomic manager needs to follow some common behaviour. In Section 3.1 we explained the general architecture of an autonomic manager including the MAPE loop, however we are dealing with a number of these managers running on different places in the hierarchy and they need to be able to communicate with each other to achieve common goals. The AMs might start or stop at

any time and therefore it is important to have mechanisms in place to detect these dynamic changes and the impact they have on the management hierarchy.

In the next Chapter we explain this general behaviour and mechanisms that each AM has to follow individually to work in collaboration with other managers. We explain specific algorithms for AM start-up, policy evaluation and termination detection and develop a communication protocol that facilitates the managers' communication and collaboration.

Chapter 5

Autonomic Manager Behaviour

This Chapter describes the general algorithms that each autonomic manager has to follow individually to make the communication and collaboration between managers in the hierarchy possible. We explain general behaviour of an autonomic manager and describe different algorithms that run inside it. These algorithms specify the process of an AM start-up, how policy evaluation is done and how termination detection is handled in the hierarchy. We also explain how to infer certain kinds of communication messages from policies.

By having a collective of autonomic managers each following these algorithms, it will be possible to build a hierarchical management system and preserve it while dynamic changes happen in the system (e.g. AMs start or stop working).

5.1 Naming Scheme

Each autonomic manager should have a name that can be resolved to an *ID* at runtime. The name and *ID* for each AM is explained in Definition 6. Separating name from *ID*, which is basically the

main access point for each AM, helps autonomic managers to dynamically register themselves in the registry and also to search for, identify and contact their parent manager in the hierarchy. An AM *ID* might change over time (e.g. as result of an IP change on system restart) but that manager will still be accessible after it updates its new *ID* in the registry (e.g. as part of the start-up process).

An AM name is dynamically provided to it as a configuration parameter by the deployment system explained in Chapter 6 and theoretically can be anything, however we propose a naming scheme in which this name includes the parent name of the manager. In other words, instead of having two configuration parameters (one for AM name and one for its parent name) we embed the parent name as part of the AM name itself. This approach is based on the assumption that each AM must be given its parent name as well to be able to look it up in the registry and start the communication process, because in the hierarchical system each manager is only able to communicate with either its children or its parent. Therefore, in this proposed naming scheme an AM name is acceptable as long as an autonomic manager can extract its parent name from it. In this section, as part of the proposed naming scheme we suggest a process of naming different AMs at different levels of the hierarchy which includes the parent name as part of it.

We suggest that an AM name depends on the location that AM is running on (e.g. machine name) plus the kind of entity that it manages. It should not be in conflict with other AM names (e.g. should not be the same) because these names should be registered

in the registry. Therefore we use a hierarchical naming convention to name AMs too. An AM name will be the name of its parent in the hierarchy plus the location and managed element name separated by a symbol (e.g. dot in this case).

AM Name = Parent Name"." (Location["-" Managed Element Name])

The location is a physical or virtual machine name that this AM is running on and the managed element name is either the name of the entity this AM is managing or is null. Therefore, the name of the root manager responsible for the whole cloud can be "host1-cloud" or simply "host1" and for a manager responsible for the first cluster of that cloud can be "host1-cloud.host2-cluster1" or if it is running on the same machine as the cloud AM it can be "host1-cloud.host1-cluster1". Similarly for the manager of a physical node in the first cluster can be "host1-cloud.host2-cluster1.host3", the managed element name is not added to the location in this name (e.g. it is null). This naming scheme helps the deployment system to automatically deploy autonomic managers to the right place and the process of dynamically creating this name is explained in Algorithm 6.2 of Chapter 6. Moreover, administrators have the ability to decide and choose the structure of the managed element name and when it should be null (through technology scripts explained in Section 6.2) based on their administrative needs and their computing environment.

In a lower level, the name of the AM responsible for managing a virtual machine can be "host1-cloud.host2-cluster1.host3.vm1" and this approach is used for all managers running inside the virtual ma-

chines. For example, the name of an AM managing Apache inside vm1 will be "host1-cloud.host2-cluster1.host3.vm1.vm1-apache". In this way, it is possible to have another AM for managing Apache on a different virtual machine with no naming conflict since the name of that manager will be "host1-cloud.host2-cluster1.host3.vm2.vm2-apache". By having a name like this, each manager has straightforward access to the name of its parent (e.g. a prefix of its own name) and it can easily use that for communication. For example, the AM with name "host1-cloud.host2-cluster1.host3.vm1" can use "host1-cloud.host2-cluster1.host3" to contact its parent in the hierarchy by looking up its *ID* in the registry.

Another challenge will be how this name can be set as a configuration parameter in an autonomic manager in a dynamic way. This happens as part of the deployment process (see Chapter 6) but the basic idea is that, at the time of deploying a new manager, one of the required steps is to configure this name based on the level on which that manager is being deployed; this is particularly relevant in IaaS. For example, when a virtual machine is being placed into a physical server, its manager should be configured with the right name which includes both the physical and virtual machine names. This process happens dynamically as part of the deployment system (see Algorithm 6.2).

In the IaaS cloud environment, this name does not change very often for most AMs, since after racks, chassis, physical servers and other elements are installed and started working in a data center they barely move around and therefore the AMs' names that get

deployed on them do not change frequently. One exception to this are virtual machines. It is possible to migrate virtual machines from one node to another either as result of a policy or manually by an administrator. If a virtual machine migrates from one node to another the manager responsible for that virtual machine and all its children should change their names to reflect this migration to a new host (e.g. a new parent in the new host is now responsible for these managers). This migration will be detected by the deployment system automatically (see Algorithm 6.1 in Chapter 6) and as a result, all AMs on the migrated virtual machines get redeployed with their new names. As part of this redeployment, AMs restart and go through the start-up process explained in Algorithm 5.1 which will result in registering the new names. The old names get invalidated in the registry by the old parent after missing to receive a response from the migrated child. This process is explained later as part of the termination detection algorithm (Algorithm 5.5).

Therefore, we propose to use a hierarchical naming scheme for different AMs and these names are configured automatically at the time of deployment by the deployment system. If there is a change in the infrastructure, the deployment system will detect that and redeploy managers automatically with their new names which will lead to a change in the management hierarchy to reflect the changes in the infrastructure. The autonomic manager deployment algorithm which includes how to determine an AM name is explained later in Algorithm 6.2.

5.2 Communication Protocol

In the hierarchical model explained before there are multiple autonomic managers deployed at different parts of the system. Each manager should be able to communicate with its parent or children and as part of this communication there needs to be a communication protocol that all managers agree on and understand.

AMs can start or stop working at any time and therefore other relevant AMs need to get notified about these changes in the system. Also, each AM works independently from others and the only means of communication is through a protocol that other AMs can understand too. AMs should not be dependent on a direct communication with each other, since they should act autonomously. Therefore, we need a way of communication that is reliable, asynchronous and loosely coupled. Considering all these facts we use a message-based means of communication between different AMs. This means that each AM can compose and send a message to another AM in a loosely coupled fashion. The message gets delivered by underlying layers but the AM does not wait for the delivery and moves on to its own operations. We assume that underlying layers guarantee that the messages will get delivered to the receiver (e.g. by using TCP and message queues). In this section, we define a communication protocol that is used between managers and explain different types of messages in detail.

Definition 8 *Given an* $AM \in AMSet$*, A message Msg is a tuple* $\langle Type, Info \rangle$ *where*

- *Type is the message type* $\in \{NOTIFY, UPDATE\_REQ, INFO\}$ *and*
- *Info* $= \{m_1, ..., m_j\}$, $\forall$ *1*$\leq k \leq$*j* $\exists$ *MEI* $\in$ *AM.MEISubSet* $\mid m_k =$ *instance of MEI.m for UPDATE_REQ and INFO types and*
- *Info* $= \{e\}$, $\exists$ *E* $\in$ *AM.ETSubSet* $\mid$ *e = instance of E, for NOTIFY messages.*

Each manager should be able to receive messages from or send messages to other managers. By using a message "Type", we introduce the possibility of different types of relationships between managers (e.g. request, response) and based on the type of message, one manager can expect the kind of information that would be available in the *Info* section of the message. Three different types of messages (NOTIFY, UPDATE_REQ, INFO) are proposed for communication between managers.

Having a small set of different types of messages also makes it easy to define the operation of each AM. The form of each of these types of messages is as follows:

- $\langle NOTIFY, Info \rangle$: When one manager wants to raise an event in another manager it can be encapsulated inside a notify message. The type and content (metrics) of the event is very system specific and can be defined in the *Info* portion of the message. Possible events would be a "HelpRequest", "SLAViolation", "SystemRestart" or "ValueUpdate" event. When a manager receives a notify message from another manager, it will publish its event and deliver it to the interested subscribers (e.g. evaluate proper policies). This type of communication message can

be either specifically declared in policies that used for a communication or it can be inferred automatically from policies (see Section 5.6).

- $\langle UPDATE\_REQ, Info \rangle$: This is a message asking for the status of the metrics declared in *Info*. Another manager can respond to this message by sending an INFO message back. The metrics are very dependent on the nature of the system and can be different from one system or application to another. Examples of such information include CPU utilization, memory utilization, number of requests/second, number of transactions, available buffer space, packets per second, etc. One usage of this message is illustrated in termination detection of child managers as explained in Algorithm 5.5.

- $\langle INFO, Info \rangle$: This message can be used to send the latest metrics of elements managed by a particular local manager to another AM. This is a message that provides information which can help the process of decision making in the higher level manager. This message is usually sent in response to the UPDATE_REQ message from a higher level manager.

The UPDATE_REQ message is sent from one AM to another to request an update of information, e.g. from higher level managers to lower level ones. INFO messages are sent in response to the UPDATE_REQ message and NOTIFY messages are sent from one manager to another based on the need.

As noted, we assume that the underlying layer guarantees message delivery. However, since communications are asynchronous, we

also assume that after sending an UPDATE_REQ message the autonomic manager waits for a configurable amount of time to receive an INFO message back. If it does not receive any response back during this time period it assumes that the other manager has terminated and raises an event inside the manager. We will explain in more detail how we can use policies to generate NOTIFY messages for communication among AMs. Since we are dealing with a hierarchy of managers, each manager needs to communicate with either its father or its children. However, it is also possible for an AM to send NOTIFY messages to another AM in some other part of the hierarchy based on a request.

5.3 Start-up

There are certain steps that need to be accomplished at each AM start-up. This start-up can happen after a successful deployment of the AM (Algorithm 6.2), after a failure/crash for any reason or upon the request of the parent manager (e.g. based on a policy). The start-up algorithm is shown in Algorithm 5.1. Each AM (see Definition 6) is provided with two pieces of information at the time of deployment: 1) AM name 2) Set of managed elements it's supposed to monitor in the form of *ManagedElementInfos* (e.g. *MEISubSet*). Therefore, an AM has access to its parent's name by extracting it from its own name (using our current naming scheme). *MEISubSet* helps the manager to load the relevant policies from the repository.

The AM identifier (AM_ID) is a globally unique identifier that gets created at AM start-up. This ID is the main access point of the AM and can be a URL or an IP address etc. based on the

actual implementation of the system. The *AMName* is configured automatically through configuration parameters for each AM by the deployment system (Algorithm 6.2).

Algorithm 5.1 AM Startup

Require: $AMName, MEISubSet$

1: $ParentName \leftarrow prefix(AMName)$
2: $AM\_ID \leftarrow$ Create unique ID (e.g. URL || IP)
3: $PolicyLocation \leftarrow Registry.register(AM\_ID, AMName)$
4: $Policies \leftarrow LoadPolicies(PolicyLocation, MEISubSet)$
5: KnowledgeBase.store($Policies$)
6: StartManagementThreads()
7: **if** $ParentName == null$ **then** ▷ No parent is configured
8: **return** ▷ Return and wait for child AMs to connect
9: **end if**
10: $ParentID \leftarrow null$
11: **while** $ParentID == null$ **do** ▷ Parent Name is configured, Beaconing starts
12: $ParentID \leftarrow Registry.getID(ParentName)$
13: Sleep($KeepAliveTimer$)
14: **end while** ▷ Beaconing ends
15: $Event \leftarrow \langle NewChildManager, \{\langle \text{“}Name\text{”}, AMName\rangle, \langle \text{“}ID\text{”}, AM\_ID\rangle\}\rangle$
16: $Msg \leftarrow \langle NOTIFY, Event\rangle$
17: SendMessageTo($ParentID, Msg$)

The autonomic manager starts by contacting the central registry (see Section 4.1) and registers its own name with the unique ID. The registry will then return the location of policy repository and the manager loads all related policies from the policy repository based on the set of *ManagedElementInfos* (e.g. *MEISubSet*) that this AM is managing (see Definition 6). For example, if a manager is responsible for monitoring a virtual machine, it loads all policies that are related to a virtual machine. Similarly, if it is responsible for monitoring Apache and MySql applications, it will only load all policies defined for these two elements. After loading these policies, they will be stored in the common knowledge base (e.g. *Knowledge-*

Base.store()) of the autonomic manager (see Section 3.1) so that other parts of the manager can access them.

It then starts other parts of this autonomic manager (e.g. *StartManagementThreads()*) such as monitoring loop (Algorithm 5.2), management interval loop (Algorithm 5.3), policy evaluation loop (Algorithm 5.4) and termination detection loop (Algorithm 5.5). We assume these algorithms will run on separate threads and since we have adopted the event driven approach they can publish new events to the system and will be notified about the events that they subscribed to receive.

In case there is no parent name set up for an AM (e,g, *ParentName == null*), it just waits for its child managers to contact it. If there is a parent name available, it will look up its parent's ID by contacting the registry and providing the parent's name. This mechanism is very similar to phone book lookup, when someone can lookup a person's phone number by having his name. In this process, when an AM gets started it knows the name of its father (through the configuration parameters set by the deployment system) and will look its ID up in the central registry and will be able to contact the parent after that.

In the case there is a parent name configured and no parent ID available in the registry, the AM will keep asking the registry in a configurable time interval (e.g. *KeepAliveTimer*) until its parent becomes available (e.g. register itself in the registry). We call this process "beaconing" when an AM beacons out and looks for a par-

ent in the management hierarchy. This case only happens if the parent AM is not deployed yet or it is terminated because of an error. In both of these cases, the periodic discovery algorithm in the deployment system (Algorithm 6.1) makes sure that the parent AM gets deployed and starts running eventually. During the beaconing period, other parts of the manager continues its job (e.g. running on separate threads) and enforces its policies based on the latest updated metrics. So, the manager continue its monitoring and management as well as periodically checking for its parent.

After a successful beaconing, the AM creates a new event (e.g. *NewChildManager* event) and add its own name and identifier to it to be sent to the parent. The manager will then wrap this event in a NOTIFY message (see Definition 8 for details of a message format) and send it to its parent which will result in getting added to the children list of the parent.

Therefore, after an AM is deployed and during its start-up it will find the right position in the hierarchy and will get added to the management hierarchy automatically. Algorithm 5.1 shows this process. After an AM starts-up it will be able to register itself and join the management hierarchy and acquire its policies from the policy repository for enforcement. We assume that there are policies defined for different elements that need to be managed and they are stored in this repository. For example, we assume that there are a set of policies for virtual machines and that those managers who are monitoring VMs will extract these policies for enforcement. There are a set of policies for host machines, applications, etc.

5.4 Processing

After a successful AM start-up, the manager needs to monitor managed elements and enforce loaded policies which are now stored in the common knowledge base and all parts of the manager have access to them. This essentially happens as part of the MAPE loop explained in section 3.1 but different algorithms are involved as part of this process. In this section we explain multiple algorithms that are running separately (e.g. on separate threads) but can communicate with each other, either through publishing events or accessing elements in the common knowledge base as shown in Figure 3.1.

Algorithm 5.2 shows the monitoring loop (e.g. M in the MAPE loop) which monitors all managed elements of this autonomic manager in a configurable time interval (e.g. *RI* - see Definition 6). It first instantiates all *MEIs* that are configured for this AM (e.g. *MEISubSet*) and stores them in the common knowledge base (e.g. *KnowledgeBase.store(MEOSet)*) so that other algorithms (e.g. threads) can access them. It will then start refreshing the metrics available in those *MEOs* in a loop on every *RI* time period. The new values are automatically updated in the common knowledge base once the refresh is done.

Algorithm 5.3 shows the management interval loop inside each autonomic manager. This loop is triggered based on a configurable management interval which can be in the milliseconds, seconds or minutes time scale. Upon each management interval (e.g. *MI* - see

Algorithm 5.2 Monitoring Loop

Require: $RI, MEISubSet$

```
1: MEOSet ← null
2: for all MEI ∈ MEISubSet do
3:     MEO ← new MEI                                  ▷ Instantiate MEI
4:     MEOSet.add(MEO)
5: end for
6: KnowledgeBase.store(MEOSet)
7: while true do
8:     for all MEO ∈ MEOSet do
9:         MEO.refreshMetrics()
10:    end for
11:    Sleep(RI)                                      ▷ Refresh Interval
12: end while
```

Definition 6) it raises a *ManagementInterval* event which will then be received and processed by the policy evaluation loop (Algorithm 5.4). Since we use event-condition-action policies the policy evaluation loop can subscribe to receive different events and this management interval loop acts as a timer that trigger an event on each *MI* to enforce relevant policies.

Algorithm 5.3 Management Interval Loop

Require: MI

```
1: while true do
2:     Sleep(MI)                                      ▷ Management Interval
3:     MgmtIntervalEvent ← ⟨ManagementInterval, null⟩  ▷ No event metrics
4:     Publish(MgmtIntervalEvent)
5: end while
```

The autonomic manager determines the set of actions that needs to be executed in each management interval (e.g. A and P in the MAPE loop) based on the latest monitored information of managed elements (e.g. *MEOSet* - see Definition 2) which are available in the common knowledge base (see Algorithm 5.2), raised events and active policies (e.g. *PLSubSet* - see Definition 6).

Algorithm 5.4 shows the policy evaluation loop that happens inside each autonomic manager. We adopted an interrupt-driven (blocking) approach towards event-condition-action policy evaluation. In this approach, the thread first subscribes to receive all events that are related to the policies (e.g. *Subscribe(ETSubSet)*) and then waits silently until new events are published in the system. This event could be a *ManagementInterval* event or a *HelpRequest* event that was sent from a child manager. *PLSubSet* and *ETSubSet* are defined in the Definition 6. *PLSubSet* is the set of policies that this manager acquired from the repository during the start-up process (Algorithm 5.1) and *ETSubSet* is the set of event types used in those policies. *MEOSet* is the set of *MEOs* available in the common knowledge base and are refreshed by Algorithm 5.2.

Algorithm 5.4 Policy Evaluation

Require: $MEOSet, ETSubSet, PLSubSet$

```
1: Subscribe(ETSubSet)                          ▷ Subscribe to receive all policy events
2: while true do
3:     Event ← Block and wait to receive new events
4:     for all Pl ∈ PLSubSet do                              ▷ Received new events
5:         if Pl.E = Event then                              ▷ Policy event is triggered
6:             for all MEO_i ∈ MEOSet do                     ▷ Latest monitored information
7:                 successful ← PolicyEngine.evaluate(Pl, MEO_i)
8:                 if successful = true then
9:                     ExecuteActions(Pl.A, MEO_i)
10:                end if
11:            end for
12:        end if
13:    end for
14: end while
```

Upon receiving new events (e.g. *Event*), the AM checks all policies one by one to see if the event in the policy matches the published

event. If for a specific policy, the event is triggered, then the policy engine matches the *MEO* values with the conditions defined in the policy (in terms of *MEI* metrics) and if the policy condition is satisfied, the AM then executes the policy actions in the same order that are defined. The *ExecuteActions* method is explained in more details in Algorithm 5.6. This process gets repeated for all active policies in the policy set *PLSubSet* and after that it blocks and wait for other events to get published.

5.5 Termination Detection

An AM might shut down at any time either because of a failure or because of a normal termination, such as when the objects that it manages terminate. When a particular AM shuts down, its parent needs to detect that in order to take necessary actions. One of these actions is to update the registry and invalidate the entry for the dead child. Another action is to raise an event which leads to enforcing those policies that are related to the termination of a child.

Algorithm 5.5 shows the process of termination detection inside a manager. This algorithm also runs on a separate thread and starts as part of the start-up algorithm (Algorithm 5.1). In order to detect this termination, each AM has a configurable keep alive timer (e.g. *KeepAliveTimer*) to check if its children are still alive or not. It will send an "UPDATE_REQ" message asking for an update and will wait for an "INFO" message to come back. If it does not get any response back, it will assume that the child is dead and will raise an event (e.g. *AMTerminationEvent*- to be evaluated in the policies)

and remove it from its *ChildrenList*.

Algorithm 5.5 AM Termination Detection and Removal

Require: ChildrenList

while *true* **do**
  for each *Child* in ChildrenList **do**
    $Metrics \leftarrow \{\langle \text{“}Name\text{”}, null\rangle, \langle \text{“}ID\text{”}, null\rangle\}$
    $Msg \leftarrow \langle UPDATE\_REQ, Metrics\rangle$
    SendMessage($Child.AM\_ID, Msg$)
    if *NoResponse* **then**
      ChildrenList.remove(*Child*)
      Registry.InvalidateEntry($Child.Name, Child.AM\_ID$)
      $Info \leftarrow \{\langle \text{“}Name\text{”}, Child.Name\rangle, \langle \text{“}ID\text{”}, Child.AM\_ID\rangle\}$
      AMTerminationEvent $\leftarrow \langle AMTermination, Info\rangle$
      Publish(AMTerminationEvent) ▷ Raise the event
    end if
  end for
  Sleep(*KeepAliveTimer*)
end while

If at a later time, the child starts communicating with this AM, it will get added to the *ChildrenList* again. When an AM dies, its parent will detect that and will raise the proper event for policy evaluation. Note that, if, for example, a virtual machine shuts down then all the applications inside that VM are also shut down and therefore all the children of the virtual machine AM are already dead. Hence, sometimes if an AM in the hierarchy terminates, the whole subtree rooted at that AM will be terminated too.

However, if one manager is terminated due to an error/fault, it is possible that its children are still running and become disconnected from the hierarchy temporarily. In this case, the deployment system will detect the termination of that AM during its periodic checking and will restart it. The AM then goes through the start-up process

(Algorithm 5.1) and gets added to the management hierarchy again. It will then be able to receive its children's messages and also send messages to its parent. This checking process is explained later in Algorithm 6.1 in Chapter 6, but the idea is that as part of periodic checking for changes in the infrastructure, the deployment system will also check the running status of the deployed managers and if they are terminated it will restart them. Therefore, for a short period of time, the subtree rooted at the terminated AM might be disconnected from the rest of the hierarchy but it will recover once the manager is restarted. However, during this temporary disconnection those managers are still working and continue enforcing their policies locally. The scaning period to dicover terminated AMs is a configurable time interval in the deployment system. However, the children of the terminated node can detect that their parent has been terminated by checking the central registry - a terminated node's name is invalidated in the registry. This algorithm is linear (O(n)) in the number (n) of child managers for this autonomic manager.

5.6 Inferring Messages From Policies

One of the challenges in collaboration between managers is to determine when they need to send/receive a message from another AM in the hierarchy. Since we are using a policy-based approach, one way to specify when a message should be sent is to have a specific policy that determines when an AM is to communicate. For example, one could include a policy explicitly identifying a communication action to send a help request event from a lower level to a higher level manager; such as:

OnEvent: ManagementInterval
if ApacheMEI.ResponseTime > 1000 **then**
    ApacheMEI.SendHelpRequestEvent = true
end if

This approach requires work by the administrators in order to define all the policies needed. An alternative would be to automatically infer from policies the right time for sending a message and the content of that message. In the remainder of this section, we explain how autonomic managers can infer certain kinds of messages; determine the right message type and the right time for sending a message to another AM.

When a manager has detected an SLA violation it tries to execute the associated corrective actions. If one of those actions fails or cannot be executed, e.g., an action to increase the value of some parameter but has reached some limit in changing that parameter, then it cannot make a local adjustment. Given the message types we have described, it will then create a NOTIFY message and send it to the higher level manager to ask for help. That is, as long as there is something that can be done locally there is no need for further communication unless it is an UPDATE_REQ message.

Algorithm 5.6 is in fact the *ExecuteActions* method in Algorithm 5.4 and shows how an autonomic manager can infer a NOTIFY communication message automatically at the time of executing actions (e.g. E in the MAPE loop). This algorithm should run during a policy evaluation and the proper *MEO* and *ActionSet* are given to it at that time. In other words, when policy conditions are met (e.g. be-

Algorithm 5.6 ExecuteActions

Require: $ActionSet, MEO$
1: $localLimitation \leftarrow false$
2: **for all** $A \in ActionSet$ **do**
3: **if** $Execute(A) == false$ **then** ▷ An action failed to execute
4: $localLimitation \leftarrow true$
5: break
6: **end if**
7: **end for**
8: **if** $localLimitation == true$ **then**
9: $Event \leftarrow \langle HelpRequest, MEO.m \rangle$ ▷ Attach latest metrics
10: $Msg \leftarrow \langle NOTIFY, Event \rangle$
11: SendMessageTo($ParentID, Msg$)
12: **end if**

cause of an SLA violation) a set of ordered corrective actions should take place to fix the stress situations. At the time of performing these actions if one of them fails due to reaching a limitation or an error, it means that local adjustments are not possible and therefore this manager should notify a higher level manager. So, after executing these actions and if there is a failure in execution of one of them, other actions will not be executed because of the "ordered" property on this set of actions (see Definition 5) and the manager will create a *HelpRequest* event and attach all of the latest metrics of the proper *MEO* to it. It then creates a NOTIFY message and sends it to its parent. Note that *ParentID* is retrieved at start-up time (e.g. Algorithm 5.1) and is available in the common knowledge base.

In order to better illustrate this process, we show several examples of policies that can be used at different levels of a hierarchy and how these policies can influence the relationship between managers. Assume that on a virtual machine there is a LAMP (Linux-Apache-

Mysql-PHP) stack that hosts web applications and that one AM is managing the applications inside this virtual machine. We use event, condition, action (ECA) policies to specify operational requirements, including requirements from SLAs, and we also use policies to identify and react to important events. Assume that the following policy is being utilized by AM_{vm1} (see Figure 5.1) and is a policy specifying the requirements needed to meet an SLA. The policy indicates that the Apache response time should not go above 500ms. This policy gets evaluated once a "ManagementInterval" event happens.

OnEvent: ManagementInterval
if ApacheMEI.ResponseTime > 500 **then**
    ApacheMEI.IncreaseMaxClients(25)
end if

This policy specifies that if the response time of the Apache server goes beyond 500ms, then the manager should increase the *MaxClients* configuration parameter (inside Apache) by 25. Now, assume that the limit for *MaxClients* is 200, which means that the manager can not increase it to more than 200. The actual actuator that performs the increase will be aware of this limitation. Therefore, upon reaching this limit there is no further local adjustment possible and the action execution will fail. This manager can then automatically create a NOTIFY message and send it to its parent manager. It basically means that after reaching the local limits of any parameters or when other actions fail each manager can ask for help from the parent manager in the hierarchy automatically. The parent manager will then receive this help request and reacts to it based on its own set of policies.

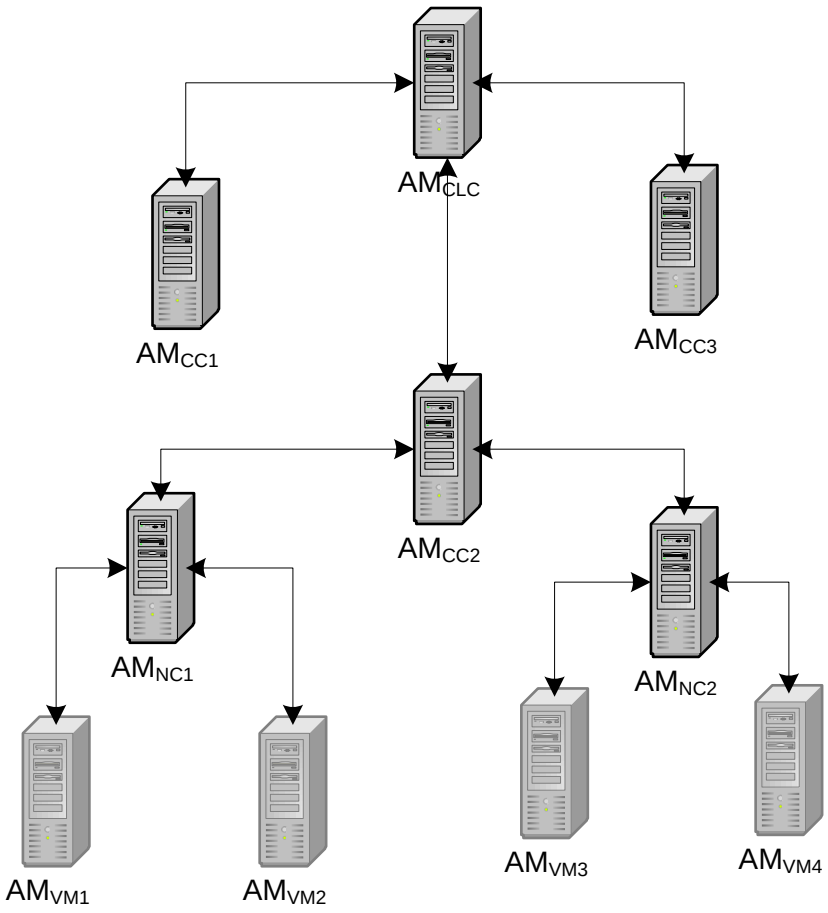

Figure 5.1: AMs hierarchy based on the cloud architecture

Upon receipt of a *HelpRequest* notify message from another AM (e.g. AM_{vm1}), a *HelpRequest* event gets triggered inside the receiving manager and those policies that match that event get evaluated by the manager. Another sample policy for an AM at the node controller level (e.g. AM_{nc1}) is:

OnEvent: HelpRequest
if VirtualMachineMEI.MemoryUtil>85 &
VirtualMachineMEI.CPUUtil>95 **then**
    VirtualMachineMEI.IncreaseVMMemory(50)
end if

This policy specifies that when a *HelpRequest* event happens, if the memory utilization of the VM in need is more than 85% and its CPU utilization is more than 95%, then the manager should increase its memory by 50 MB to help the stressed virtual machine. Again, this can only be done to some limit. In this case, the maximum limit can be determined at runtime by the actuator responsible for increasing the memory.

Therefore, if there is no extra memory available to be added to the stressed virtual machine, the actuator will fail, the action fails and the manager will automatically create a NOTIFY message to be sent to its parent in the hierarchy. In this case, it is basically notifying its parent about a stress situation that can not be resolved locally and asks for more help. This process is implemented and illustrated in more detail in Section 7.1 of Chapter 7.

In this way, an AM can determine exactly when it has reached the local limits and create the NOTIFY message to be sent to the higher level manager. Thus, the AM can infer automatically from the policy when to send this type of message at run time. This happens when the manager has reached the local limits in trying to enforce actions specified in a policy. However, the need to have another policy at the higher level manager to respond to these events is not removed. The process of creating/inferring and sending of the messages from one manager to another is automated but there still should be another policy that react to these incoming events (e.g. NOTIFY messages). Since there are a lot of possible ways to define these policies and it is different from one organization to another, we assume that administrators of each organization should define these policies based on their needs. The UPDATE_REQ and INFO messages are used to maintain the hierarchy and detect the termination of an AM as explained in Section 5.5.

Based on this technique we can build a system with different AMs working autonomously at different levels and interacting with each other based on demand but the important point is that all these AMs are collectively trying to adhere to a set of policies that minimize the number of SLA violations (or maximize performance based on SLA parameters), and minimize resource usage at the same time. This happens while each manager has a local view of the system and is trying to solve problems locally but when no further local adjustment is possible it asks the higher level manager for help.

Chapter 6

Autonomic Manager Deployment

In large systems there are multiple elements that need to be managed. For example, in a typical data center there are elements like clusters, physical nodes, virtual machines and applications running on them. Some of these elements can get added to or removed from the system dynamically and therefore the management should be able to adapt to these changes.

The deployment of autonomic managers at different parts of the system can be a challenging task for administrators as the number of elements grow. Deployment in our hierarchical management context means installing an AM with correct set of *ManagedElementInfos* (e.g. *MEISubSet*) in the right place and, in our case using our naming scheme, configuring the correct AM name which is prefixed with the parent name in the hierarchy. We introduce a deployment system that can automate the autonomic manager deployment process and which can keep the management hierarchy up to date as changes happen in the infrastructure.

There are two possible ways of autonomic manager deployment:

1)Manual: which is useful for monitoring and management of custom applications running inside virtual machines and are not already included in the VM image and 2) Automatic: which happens as part of the deployment system explained here. For manual autonomic manager deployment, an administrator can install the AM and configure its name and therefore has to know the management hierarchy structure and the parent name of the AM to be installed. This process is similar to the traditional way of installing AMs and configuring them by an administrator. The naming scheme we introduced earlier can help the administrator create these names. For automatic deployment, the deployment system needs to be able to create the AM name on the fly based on the location of that AM and therefore needs to have access to the list of different elements' names in the computing environment. For example, depending on the naming convention being used and in order to deploy an AM in a virtual machine, the deployment system needs to know the virtual machine's name, its host physical machine's name and the cluster's name that it is running on in order to be able to create a name for that AM.

The deployment system consists of several management tables and each one holds some information that facilitates the automatic deployment process. In this section we explain details of these tables and how each table should be filled. Throughout this section we assume that the right credentials are put in place and the deployment system has authorized access to the physical nodes and virtual machines located in the computing environment. In the rest of this Chapter, we explain how a deployment system can be created to automatically deploy AMs in the right position with the right name

and make sure that these AMs are running.

6.1 Management Groups

To be able to deploy AMs that are organized in a hierarchical fashion the deployment system first needs to know what management groups exist in the system and their relation to one another. These management groups can correspond, for example, to the levels of the hierarchy common in an IaaS cloud, e.g., application, virtual machine, compute node, etc. Since we focus on a hierarchical organization of AMs, we assume that the management groups are defined as a tree structure; the groups are defined as Management Groups (MG) and get added to the deployment system manually by administrators.

Each management group is associated with one level of the hierarchy and represents a group of elements that should be managed in the same way (e.g. have the same policies). There is no limit on the number of levels in the management tree or the number of management groups at each level. Administrators of any computing environment can decide on the number of levels and how to organize their infrastructure into a hierarchical structure. These management groups are stored in the *MG* table. Table 6.1 shows an example of a management group table and the level column shows the level of each group. We assume that Level 0 is the root level, level 1 is one level below with all root's children and so on. Another example of this table in IaaS clouds is explained in Section 6.6.

Table 6.1: Management Groups

Level	MG Name
0	MG1
1	MG2
2	MG3
2	MG4
3	MG5
⋮	⋮

In order to support different possible management approaches at each level, one can define multiple management groups for each level of the management hierarchy. For example, there might be multiple nodes using different hardware at the node level and therefore administrators need to manage them differently. They can define multiple management groups in this table for that specific management level (e.g. level 2 in Table 6.1). By defining different management groups, we can now assign multiple attributes to each group in order to facilitate the AM deployment for each level of the hierarchy.

6.2 Management Group Attributes

All attributes that belong to a certain management group are added to another table. Note that this table and the previous table (e.g. *MG* table) can both be combined into one table. However, for explanatory reasons we split them into two tables to explain elements more clearly.

One of the main attributes that has to be assigned to each group is the set of elements that need to be managed in that group (e.g. *MEISubSet*). Administrators have to specify what elements are im-

portant to them to be monitored in each group by adding the desired *ManagedElemmentInfo* to the correct group. This can be done in *MGAttributes* table. This table helps the deployment system to determine which *MEIs* should be included with a specific AM deployment. For example, when deploying an AM responsible for a cluster it might not be necessary to include an *ApacheMEI* or when deploying an AM responsible for a virtual machine, some of the application *MEIs* (e.g. *MySqlMEI*, *ApacheMEI*, etc.) as well as *VirtualMachineMEI* might be included in the deployment process.

Another attribute is the name of a technology script that is used at that level of the hierarchy. This could be the name of a predefined supported script that is included with the deployment system (in the form of executable scripts) or any new script that administrator add to the system based on their computing environment technology stack. For example, administrators can specify a script name that is used at the Eucalyptus cloud level (e.g. management group). The script name specified in this table is in fact the path and name of an executable script. The scripts can also be stored on a specific folder under the deployment system file structure to avoid including path for each one (e.g. "/opt/deployment/scripts/"). This script helps the deployment system to extract proper information (e.g. children names) from each level of the management hierarchy. For example, in order to get access to the physical machine names at a cluster level (part of discovery process explained in Algorithm 6.1), the deployment system needs to know which technology is used in order to get these names from the right place (e.g. a MySQL DB, a specific file, etc.) and these scripts will help the deployment system to extract

this information.

These scripts are extensible and can be added or modified to support more technologies in the deployment system based on what infrastructure this deployment system will be run on. The main jobs of each script are 1) Check to see if a technology (or multiple technologies) is available on a specific machine or not (e.g. the technology is properly installed on it) and 2) Extract the children names based on the logic that administrators have provided inside the script and in compliance with the naming scheme explained in Section 5.1 (e.g. "Location-Managed Element Name"). For example, there might be a shell script named "Eucalyptus-0.sh" that provides the functionality to check if a specific machine has Eucalyptus cloud controller installed on it or not and also can extract the children machine names (e.g. machines with Eucalyptus cluster controller) from the Eucalyptus database based on the logic that is provided in that script. Another shell script name might be "KVM.sh" which can determine if KVM virtualization is available on a machine or not and can extract the children names (e.g. virtual machine names inside that host) based on the logic provided in that script (e.g. by running KVM commands). These scripts can be written in different scripting programming languages such as Python, Ruby, shell commands, etc. Appendix B shows two examples of these scripts in Ruby programming language.

In this way, administrators can add, remove or modify these scripts to ensure that the deployment system supports proper technologies and fits with their requirements. After defining and adding

Table 6.2: Management Group Attributes

MG Name	MEI(s)	Technology Script Name
MG1	MEI1	T1.sh
MG2	MEI2,MEI3	T2.py
MG3	MEI4,MEI5,MIE6	T3.rb
⋮	⋮	⋮

Table 6.3: Management Groups Members

Name	MG Name	Parent Name
Host1	MG1	null
Host2	MG2	Host1
VM1	MG3	Host2
⋮	⋮	⋮

these technology scripts, their names can be used in this table inside the *Technology Script Name* field. Table 6.2 shows an example of *MGAttributes* table. For example, *MG1* has only one element to be monitored (*ME1*) and the script used at this level is a shell script named "T1.sh". Another example of this table in IaaS cloud environments is provided in Section 6.6.

6.3 Management Group Members

Each management group has members that are candidates for an AM deployment at that level of the management hierarchy. Another table, the *Members* table, is used for storing information related to these members of management groups. Table 6.3 shows an example of the *Members* table.

Each member belongs to a management group and represents a location (e.g. virtual or physical machine) that an autonomic man-

ager should be deployed on and the names used in this table will be used to configure the AM names at the time of deployment based on the naming scheme explained in Section 5.1. Each member can have child members associated with it which form the hierarchical structure of the computing environment. The *Name* field in this table is a name that uniquely specifies a machine (either virtual or physical) to deploy an AM on it and the *Parent Name* field represents the member name of its parent. Names added to this table have to be unique but they do not necessarily have to be only a machine name. For example, a name can be "VM1" which is only a virtual machine name and uniquely identifies that machine or it can be "VM1-Apache" which also uniquely specify the same virtual machine but is a different name than the first one (e.g. are not in conflict). Having names that uniquely specify a machine and are different (e.g. unique) at the same time will help us construct AM names automatically (Algorithm 6) at the time of deployment based on the naming scheme explained in Section 5.1.

The first entry to this table is the root member of the management hierarchy which belongs to the first management group (e.g. level 0) and must be added by administrators manually. However, the other members (e.g table entries) can be discovered automatically by the deployment system and the help of scripts in *MGAttributes* table. Therefore, script writers (e.g. administrators) can decide how to name different elements in their computing environment as long as these names uniquely identify a machine and is not in conflict with other names. The deployment system uses this table to discover other members automatically by starting from the root

member. This discovery process is explained in Algorithm 6.1. This is a very important table because each entry that gets added to this table means that a new AM deployment should take place and if an entry is about to be deleted from this table, it means that a previous deployment is no longer valid and therefore all children of that entry should be deleted too.

Another key aspect of this table is that it represents the layout of the computing environment and therefore has to be kept in sync with the actual infrastructure layout (both physical and application-level layout). This means that any changes in the infrastructure has to be detected and get updated in this table. For example, if a virtual machine is no longer running on a physical host (either because of a migration or termination) then its entry to this table and all its children has to be deleted to keep this table in sync with the actual environment. This process of keeping this table up to date happens as part of the discovery algorithm by a polling mechanism that is explained in more details in Algorithm 6.1.

6.4 Discovery Algorithm

So far we have defined tables that hold important basic information about the computing environment. We can now use this basic information to automatically extract other information or discover members in the infrastructure and start the deployment process. Discovery of available members in the system is the heart of this deployment system because it reduces the burden of defining all members manually and helps administrators to automate the deployment

process.

The discovery process happens periodically in a configurable time interval to ensure that all dynamic changes that happen in the infrastructure from time to time will be detected. It starts from the root member and discovers all children members. Upon discovering a new member/child, it adds the new member to the *Members* table which then leads to a new AM deployment on that member. Similarly, if it detects that an existing member is no longer available it will remove it from the *Members* table which results in removing all child members associated with it.

Algorithm 6.1 shows the discovery process. It performs the discovery at every *DiscoveryInterval* which is a configurable time interval and the overall process is a breatdh-first-search (BFS) of the computing environment. It starts from the root member which belongs to the first management group (*MG.getFirstMG()* returns the first Management Group and *Members.getMembers()* returns the members that belong to a specific MG) and uses a queue to discover new members. It starts by checking the status of the autonomic manager deployed on the current member (*isAMDeployed()*) checks the AM deployment status on a specific member). If there is an autonomic manager already deployed on that member then it should be started. The *StartAM* method starts the AM if it is not running already, otherwise it does not do anything. This status checking acts as a heartbeat polling mechanism to make sure that an AM is not terminated due to a fault or error. If an AM is terminated for any reason, then the discovery process ensures that it gets started again.

Algorithm 6.1 Member Discovery

Require: $Members, MG, MGAttributes, DiscoveryInterval$

```
1: while true do
2:     Root ← Members.getMembers(MG.getFirstMG())
3:     Queue.enqueue(Root)
4:     while !Queue.isEmpty() do        ▷ There are more members to discover
5:         CurrMember ← Queue.dequeue()
6:         if isAMDeployed(CurrMember) then        ▷ Check AM status
7:             StartAM(CurrMember)
8:         end if
9:         CurrChildSet ← Members.getChildren(CurrMember)
10:        CurrMG ← Members.getMG(CurrMember)
11:        TechScript ← MGAttributes.getTechnology(CurrMG)
12:        DiscoveredChildSet ← DiscoverChildren(CurrMember, TechScript)
13:        AddSet ← DiscoveredChildSet − CurrChildSet
14:        RemoveSet ← CurrChildSet − DiscoveredChildSet
15:        Members.removeMembers(RemoveSet)        ▷ Update Members table
16:        NextMGSet ← MG.getNextMG(CurrMG)
17:        for all MG ∈ NextMGSet do
18:            MGTechScript ← MGAttributes.getTechnology(MG)
19:            for all member ∈ AddSet do
20:                if isValid(member, MGTechScript) then
21:                    Members.addMember(member, MG, CurrMember)
22:                    AddSet.remove(member)
23:                end if
24:            end for
25:        end for
26:        Queue.enqueue(Members.getChildren(CurrMember))
27:    end while
28:    Sleep(DiscoveryInterval)
29: end while
```

The discovery algorithm then gets the set of all current available children of that member (e.g. *Members.getChildren()* method) which is stored in the *Members* table (e.g. from previous discoveries) and store it in *CurrChildSet*. This set can be calculated by a simple search on the parent name field of the *Members* table but it might be outdated due to the changes that happened in the environment from the last discovery and therefore it might not represent the most recent infrastructure layout. This is the current view of the infrastructure in the deployment system.

It then gets the management group which that member belongs to and retrieves the technology script that is used on that management group (e.g. *MGAttributes.getTechnology(CurrMG)* method). At this point, it can connect to this specific member and perform a discovery based on the technology script that is provided (e.g. *DiscoverChildren()* method). The script will then retrieve a set of children names and return it to be stored in the *DiscoveredChildSet*. *CurrChildSet* represents the current view of the deployment system about the infrastructure and *DiscoveredChildSet* represents the actual most recent infrastructure layout. Therefore, the two relative complements of these two sets give us two other sets: one is the set of members that are newly discovered and should be added to the system (e.g. *AddSet)* and one is the set of members that are no longer available and should be removed from the system (e.g. *RemoveSet*).

So far, the discovery process has calculated two separate lists that need to be updated in the system. It can then update the

Members table based on these two sets. It first removes the obsolete members from the table (e.g. *Members.removeMembers(RemoveSet)* method) and then calculates the next management group (e.g. *getNextMG(CurrMG)* method) and since there might be multiple management groups for each level, it will return a set of management groups for the next level (e.g. *NextMGSet*). It now needs to determine the new members to be added belong to which management group. This can be done with the help of the technology script. One job of the technology script is to provide functionality for determining that if the technology is available on a specific member or not. Therefore, for each management group in the next level, it will get the associated technology script name (e.g. *MGAttributes.getTechnology(MG)*) from the *MGAttributes* table and check to see if that technology is available on the newly discovered members (e.g. *isValid(member, MGTechScript)*). If the technology is available then this member belongs to that management group and can be added to the *Members* table. The *addMember* method gets a member to be added, the management group (e.g. *MG*) that it belongs to and its parent member (e.g. *CurrMember*) and add this member to the *Members* table. After adding a member to the table it will be removed from the *AddSet* and the loop continue for other members. The algorithm for adding or removing a member from the *Members* table is explained in the next section. After updating the table, it then retrieve the final list of children (most recent) for this member and add them to the queue for further discovery (e.g. in the lower levels) based on the same BFS approach.

This discovery algorithm should run from a machine inside the

administrative domain that has proper access (administrative privileges) to all members, we assume this is happening in a trusted administrative network with proper authorizations in place.

The time complexity of this algorithm is O(n^2) in the worst case where n is the number of members in the *Members* table and is equivalent to the number of AMs in the management hierarchy because it performs a BFS of the computing environment (O(n)) and on each iteration of the while loop it removes the obsolete members (*Members.removeMembers() method*) which is another BFS of a subset of the *Members* table (O(n) in worst case).

6.5 Deployment Algorithms

As explained in the previous section the output of the discovery process is basically two sets, one is the set of members that have to be added to the *Members* table (new discoveries) and one is the set of members that have to be removed from this table (not available any longer). In this section we explain what should happen after adding an entry to or removing it from the *Members* table.

Each addition to the *Members* table means discovering a new member and therefore a new AM deployment has to be performed on it. In order to perform an AM deployment, the deployment system needs to calculate two pieces of information: 1) The set of *MEIs* that has to be installed with the AM and 2) the AM name which includes the parent name as a prefix (See Section 5.1). Algorithm 6.2 shows this process and how to calculate this information on the

fly upon adding a new entry to the members table. This Algorithm is called upon each addition to the *Members* table.

Algorithm 6.2 Members Addition: Autonomic Manager Deployment

Require: $MemberName, Members, MGAttributes$

1: $CurrMG \leftarrow Members.getMG(MemberName)$
2: $MEISubSet \leftarrow MGAttributes.getMEIs(CurrMG)$
3: $AMName \leftarrow MemberName$
4: $ParenName \leftarrow Members.getParentName(MemberName)$
5: **while** $ParentName \neq NULL$ **do** ▷ Not reached root member
6: $AMName \leftarrow ParentName + \text{“.”} + AMName$
7: $ParenName \leftarrow Members.getParentName(ParenName)$
8: **end while**
9: $DeployAM(MemberName, AMName, MEISubSet)$

During the AM deployment to a new member, the deployment system can retrieve the appropriate *MEIs* by accessing the member's management group. The *MGAttributes* table can return the set of *MEIs* that has to be shipped with this deployment. The parent name however has to be added step by step by searching through the *Members* table. It starts by adding the current member name to the AM name and then while it has not reached the root member it keeps adding the parent name of each member to the AM name. Basically, it starts from the leaf of the management tree and goes one step up the hierarchy each time and adds each parent's name until it reaches the root. The parent name is accessible in the Members table and is set to null for the root member. The *getParentName* method returns the parent name of a specific member from the Members table.

After extracting these two sets of information, the deployment system can then access that member and deploy an autonomic manager on it by shipping the right *MEISubSet* and configuring the right

name which will eventually result in getting added to the management hierarchy through the start-up algorithm (Algorithm 5.1).

The time complexity of this algorithm is O(log(n)) where n is the number of members in the *Members* table because it basically traverses the height of the management tree from the leaf to the root, one step at a time.

Another change in the *Members* table happens when removing an entry from it. Removing a member means that this member is no longer available in the computing environment as the child of a specific member and therefore itself and all sub-tree rooted at this member have to be removed. For example, if a physical server is no longer available in the list of a cluster's machines, then both the server and all its previously discovered virtual machines inside it are no longer available and have to be removed from the *Members* table.

Algorithm 6.3 Members Removal

Require: $MemberName, Members$
1: $Queue.enqueue(MemberName)$
2: **while** !$Queue.isEmpty()$ **do** ▷ There are more members to remove
3: $CurrentMember \leftarrow Queue.dequeue()$
4: $ChildrenSet \leftarrow Members.getChildren(CurrentMember)$
5: **for all** $child \in ChildrenSet$ **do**
6: $Queue.enqueue(child)$ ▷ Will be removed
7: **end for**
8: $Members.remove(CurrentMember)$
9: **end while**

Algorithm 6.3 shows the process of removing an entry from *Members* table. This algorithm uses a queue and performs a breadth-first-search of the sub-tree rooted at the member to be removed and

remove all these members.

For each member to be removed (e.g. *CurrentMember*), it first get the set of its children from the *Members* table (e.g. *Members.getChildren(CurrentMember)*) and add them to the queue and then remove the member itself from the *Members* table (e.g. *Members.remove(CurrentMember)*). It then repeat this process until there is no further member in the queue for removal.

This algorithm is a BFS of a subset of the members in *Members* table. Therefore, the time complexity of this algorithm in the worst case is O(n) where n is the number of members in this table.

6.6 Deployment in IaaS Clouds

We first explain operations in a typical IaaS cloud to have a better understanding of how the deployment system will integrate with this environment. To set up an IaaS cloud, racks and physical servers are installed in a data center. Then a host operating system will be installed on these machines which will later host virtual machines. In a typical IaaS architecture (see Section 3.2), the cloud controller is installed first, along with a cluster controller which is responsible for all physical nodes in that cluster. After that, the cluster controller gets a list of active physical nodes either by discovering them automatically or by administrators who have to define them manually.

After a successful setup, cloud users can request one or more virtual machines from cloud controller. Cloud controller will then

chooses the cluster and sends the request to its cluster controller. The cluster controller chooses the nodes that should host these virtual machines and places them on to those nodes. As part of this virtual machine placement, it will configure the virtual machine with the right customized information (e.g. credentials, application and services, etc.) and then the user can connect to the virtual machine and use it. Therefore, cluster controllers have access to the names of physical machines that are available in that cluster. Physical machines have similar access to the virtual machine names running on them. These lists can change over time and new servers added to or removed from the system.

6.6.1 Sample IaaS Layout

In this section we explain the physical layout of a sample IaaS cloud and explain how our deployment system can integrate with it to discover members and deploy autonomic managers. We assume that this cloud consist of two clusters, each with two physical servers to host virtual machines. Figure 6.1 shows the physical layout of this cloud. There are two racks, each have four physical servers with an operating system (e.g. Ubuntu) installed on them. *Host1* is where cloud controller is installed and the two cluster controllers are installed in *Host2* and *Host6*. Let us assume that we use Eucalyptus technology to create this cloud and that each cluster is using a different virtualization technology (e.g. KVM or Xen). Hosts in the first cluster (e.g. *Host3* and *Host4*) are using KVM and hosts in the second cluster (e.g. *Host7* and *Host8*) are using Xen.

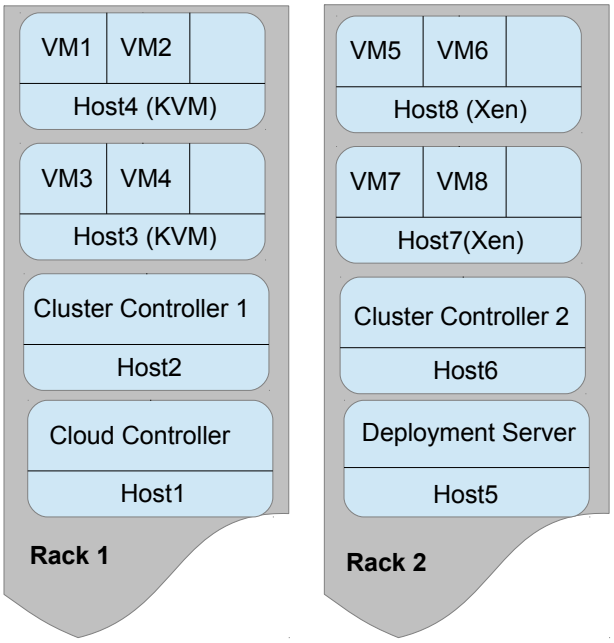

Figure 6.1: IaaS Cloud Layout

6.6.2 Deployment Tables

In order to setup the autonomic management deployment system for this sample cloud, there are several possible ways to define the management groups and form the hierarchy and depending on the actual management requirements administrators can pick one that matches their needs better. However for illustration purposes let us assume that the administrators define four management levels and five management groups for this computing environment:

1. "CloudMG": This group represents the members in the top level of the hierarchy (e.g. level 0). This group will have only one member which is the cloud controller machine (e.g. *Host1*).

Table 6.4: IaaS Cloud Management Groups

Level	MG Name
0	CloudMG
1	ClusterMG
2	KVMServerMG
2	XenServerMG
3	VirtualMachineMG

2. "ClusterMG": This group represents members running at the cluster controller level (e.g. *Host2* and *Host6*).

3. "KVMServerMG": This group represents all physical machines that uses KVM as their virtualization (e.g. *Host3* and *Host4*).

4. "XenServerMG": This group represents all physical machines that uses the Xen as their virtualizations and should be managed differently than *KVMServerMG* members (e.g. *Host7* and *Host8*).

5. "VirtualMachineMG": which represents all virtual machine available in the cloud.

All deployment tables are stored in *Host5* located in the second rack and the deployment algorithms will run from this machine. Administrators can now define these management groups by simply adding their names to the management groups table. Table 6.4 shows the *MG* table for this cloud.

Note that both *KVMServerMG* and *XenServerMG* management groups are defined to be in the second level of the hierarchy (e.g. level 2).

Table 6.5: IaaS Cloud MGAttributes

MG Name	MEI(s)	Technology Script Name
CloudMG	CloudMEI	Eucalyptus0.py
ClusterMG	ClusterMEI	Eucalyptus1.sh
KVMServerMG	KVMPhysicalMachineMEI	KVM.rb
XenServerMG	XenPhysicalMachineMEI	Xen.sh
VirtualMachineMG	VirtualMachineMEI, ApacheMEI	Ubuntu.rb

After defining management groups, administrators can add proper *ManagedElementInfos* to the *MGAttributes* table based on what elements needs to be managed at each level of the hierarchy in cloud.

Table 6.5 shows an example of *MGAttributes* table for this cloud. Based on available *MEIs* and technologies in the deployment system they can define multiple entries in to this table. For example, the Eucalyptus technology is used at the *CloudMG* level and the autonomic manager should monitor cloud level elements based on the *CloudMEI*. *Eucalyptus0.py* is the name of a Python script that will extract the children names (e.g. Eucalyptus cluster controller machine names) from the Eucalyptus cloud controller (level 0). Similarly, KVM is used at the *KVMServerMG* level and members of this group should monitor and manage physical servers based on the *KVMPhysicalMachineMEI*. The *KVM.rb* is the name of a Ruby script that can extract children names (e.g. virtual machine names) from a KVM virtualized server.

The last step in configuring the deployment system is to add the first entry of *Members* table, which is the root member of the hierarchy and is a member of the first management group. Table 6.6 shows an example of the *Members* table for this cloud. *Host1* is the root machine that cloud controller is installed on and can be accessed

Table 6.6: Initial Iaas Members Table

Name	MG Name	Parent Name
Host1	CloudMG	null

from the deployment server.

At this point the configuration of the deployment system is completed and administrators can run this service. After running the deployment service, members discovery loop (see Algorithm 6.1) starts running and the deployment process starts based on the Algorithm 6.2 and upon discovering new members they get added to this *Members* table which causes other AM deployments to take place.

As explained in the Algorithm 6.1, deployment server can now connect to the root member, extract children names by using the technology script and add them to the *Members* table. It will then continue this process until there is no member available in the queue.

Table 6.7 shows the completed *Members* table after running the discovery algorithm. Note that the discovery algorithm runs periodically to detect dynamic changes in the environment and update the *Members* table accordingly. It also checks the running status of the currently deployed AMs to make sure that they are running.

6.6.3 Deployed Managers

As described before in Algorithm 6.2, upon adding a new member in to *Members* table a new autonomic manager deployment happens. This algorithm will calculate the AM name based on the naming

Table 6.7: Completed IaaS Members Table

Name	MG Name	Parent Name
Host1	CloudMG	null
Host2	ClusterMG	Host1
Host6	ClusterMG	Host1
Host3	KVMServerMG	Host2
Host4	KVMServerMG	Host2
Host7	XenServerMG	Host6
Host8	XenServerMG	Host6
VM3	VirtualMachineMG	Host3
VM4	VirtualMachineMG	Host3
VM1	VirtualMachineMG	Host4
VM2	VirtualMachineMG	Host4
VM7	VirtualMachineMG	Host7
VM8	VirtualMachineMG	Host7
VM5	VirtualMachineMG	Host8
VM6	VirtualMachineMG	Host8

scheme explained in Section 5.1 and deploy an autonomic manager on this member with proper *MEISet* and name.

Table 6.8 shows the AM names calculated for these managers based on members available in the *Members* table. The first column shows the member name that this AM is deployed on and the second column is the AM name configured for that AM.

Upon each AM start-up, it will register its name in the registry and contact its parent which will result in forming the management hierarchy. It also gets the proper policies from the repository based its *MEISet* and starts enforcing them. Figure 6.2 shows the management hierarchy of these AMs after it starts working in our sample cloud.

Table 6.8: Deployed AM Names

Member Name	AM Name
Host1	Host1
Host2	Host1.Host2
Host6	Host1.Host6
Host3	Host1.Host2.Host3
Host4	Host1.Host2.Host4
Host7	Host1.Host6.Host7
Host8	Host1.Host6.Host8
VM3	Host1.Host2.Host3.VM3
VM4	Host1.Host2.Host3.VM4
VM1	Host1.Host2.Host4.VM1
VM2	Host1.Host2.Host4.VM2
VM7	Host1.Host6.Host7.VM7
VM8	Host1.Host6.Host7.VM8
VM5	Host1.Host6.Host8.VM5
VM6	Host1.Host6.Host8.VM6

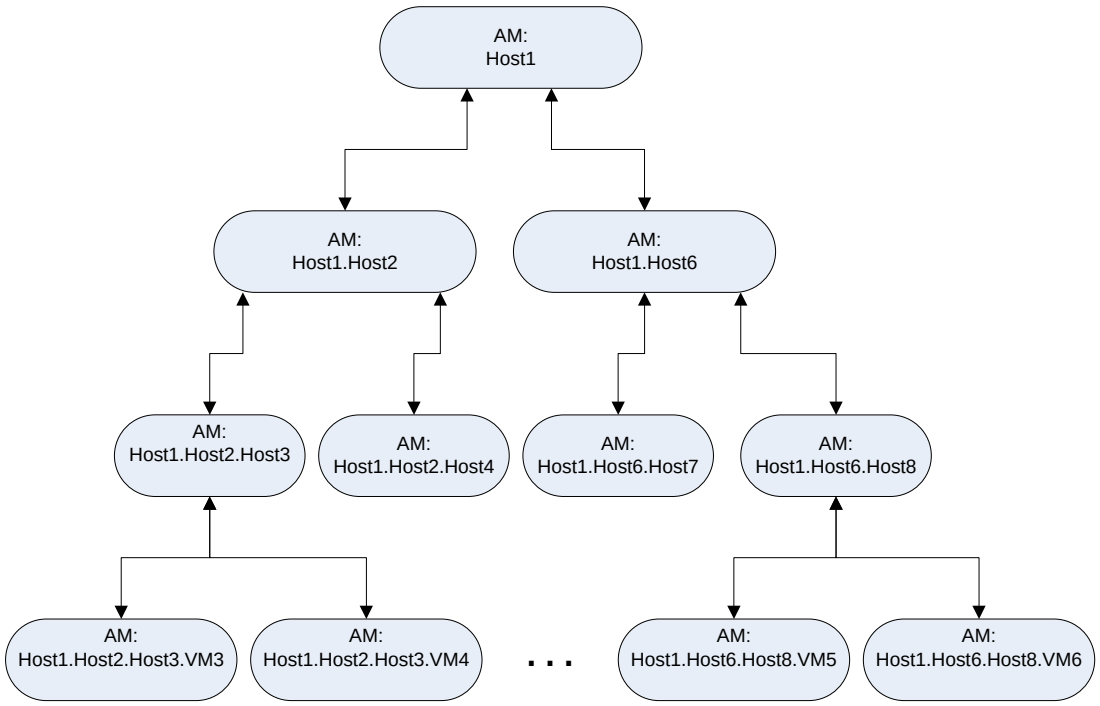

Figure 6.2: AMs hierarchy after deployment on IaaS cloud

Chapter 7

Experiments and Evaluation

In order to test our ideas, we evaluated elements of the hierarchical management model proposed in the previous Chapters in different settings. In the first setting, we used a private cloud environment and performed several experiments to evaluate the hierarchical model. In the second setting, we collaborated with a private company to address their real life management issues and implemented portions of our management model to manage their infrastructure which shows a successful application of our ideas in practice. Section 7.1 explains the details of a prototype cloud system and presents an evaluation of the hierarchical model by implementing the communication protocol, autonomic manager algorithms, such as the message inference, processing, event-condition-action policy evaluation. Section 7.2 explains the architecture and implementation details of the case study in which the ideas related to a central policy repository, registry techniques, start-up and termination detection algorithms were evaluated.

7.1 Evaluation: Performance Study

We performed experiments to evaluate the autonomic manager algorithms explained in Chapter 5, including the collaboration between different autonomic managers using the communication protocol and including the message inference from policies. We implemented a small experimental cloud environment and developed a hierarchical management system based on our approach and algorithms and measured an application's performance - in this case, Apache's response time against a service level agreement that was defined in the policies. We measured the number of SLA violations that happened during the experimental period in three different scenarios.

7.1.1 Experimental Setup

We built a small cloud with three servers, each server has 4GB of memory with Intel core i7 CPU @ 3.4GHz (4 cores) and is connected to a 10/100Mbps switch with a 100Mbps CAT5 Ethernet cable.

All servers are running a 64bit Ubuntu 11.04 and two of these servers are configured to be able to host virtual machines (VMs) using KVM virtualization [21]. We used Ubuntu Enterprise Cloud software to build this cloud which is powered by Eucalyptus (see Section 3.2). All VMs within a server can be monitored and managed from a privileged VM (e.g. Domain 0).

There are applications, e.g. an Apache web server, a MySql database server, running on the VMs. The privileged autonomic

manager runs in the physical server and its job is to manage (optimize based on policies) the behaviour of that server by collaborating with the managers running inside each VM. We used two VMs running on a single server with LAMP (Linux-Apache-Mysql-PHP) installed on them and a two tier web application based on an online store was configured to run on the VMs.

Guest virtual machines run Ubuntu 11.04 as well. "Domain 0" is the first operating system that boots automatically and has special management privileges with direct access to all physical hardware by default. The manager running inside Domain 0 has the authority to change the configuration of other VMs, such as allocated memory, allocated CPU cores, etc. Figure 7.1 shows the physical structure of the system. Server1 hosts two VMs each running a web application that receive loads. We implemented the autonomic manager using Java programming language and the Ponder2 [52] platform and used Ponder Talk to implement communication between managers.

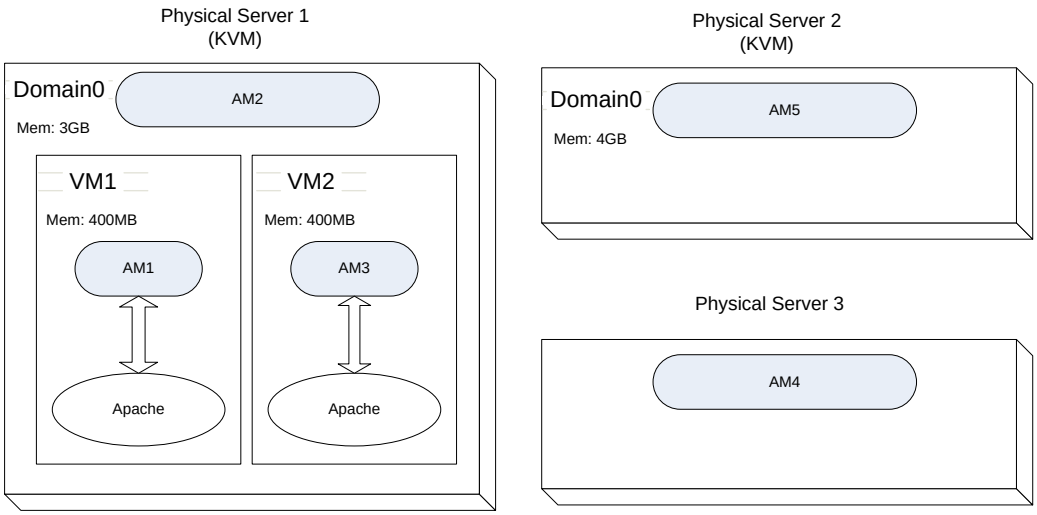

Figure 7.1: Experiments Cloud Physical layout

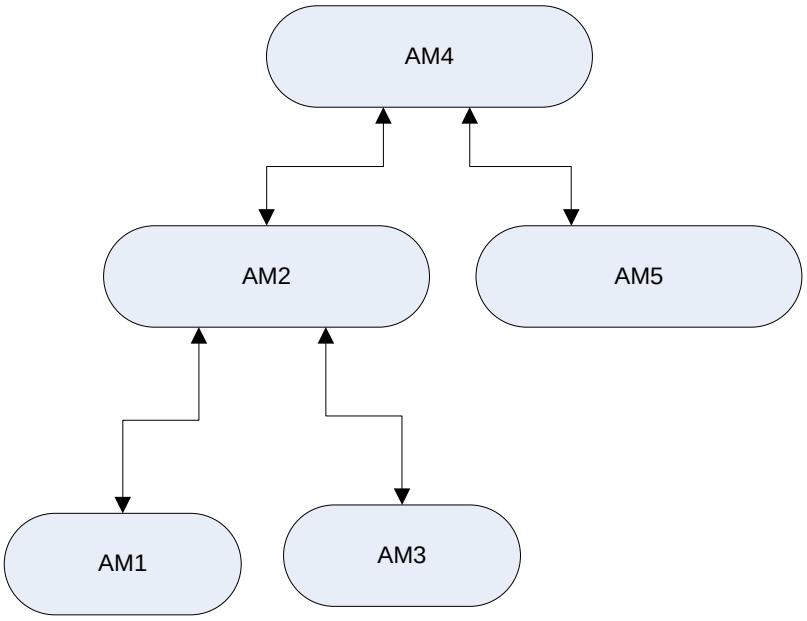

Figure 7.2: Hierarchy of managers based on physical layout

We used an open source online store called "Virtuemart" [2] as the web application, a three tier application with a Model-View-Controller (MVC) architecture, to measure the response time of Apache web server running on VM1. We also used JMeter [1] to generate loads to this virtual store and measured the response time of Apache in three scenarios. We used two thread groups in JMeter for generating HTTP requests to the online store. The first thread group is configured to have 23 threads (representing users) with a ramp up period of 55 seconds and loop count of 9. The ramp up period tells JMeter how long it should take to run the full number of threads chosen, e.g. If 100 threads are used, and the ramp-up period is 1000 seconds, then JMeter will take 1000 seconds to get all 100 threads up and running. The loop count is the number of times this test (e.g. thread group) should be repeated. The second thread group has 20 threads with a ramp up period of 10 seconds and loop count of 6. We used the same configuration for all test scenarios.

Table 7.1: Experiment's Management Groups

Level	MG Name
0	ClusterMG
1	PhysicalMahcineMG
2	VirtualMachineMG

Table 7.2: Experiment's MGAttributes

MG Name	MEI(s)	Technology Script Name
ClusterMG	NodeMEI	Eucalyptus.rb
PhysicalMahcineMG	VirtualMachineMEI	KVM.rb
VirtualMachineMG	ApacheMEI	Ubuntu.rb

The ultimate goal of the whole system is to keep the response time under a certain threshold (e.g. 500 ms) that we assumed was defined in an SLA.

There are three different management groups for this cloud (see Chapter 6) that are shown in Table 7.1 and at each level there are certain *MEIs* (see Definition 1) that get deployed with the autonomic manager. Table 7.2 shows the attributes of each management group. Appendix B shows two technology scripts written in Ruby programming language."KVM.rb" is the name of a script that checks if the machine has KVM virtualization or not (e.g. by checking "virsh" command) and it can also provide the virtual machine names as the list of its children. As it is shown in the scripts, "Ubuntu.rb" return an empty list of children because this is used at the last level of our hierarchy and there is no child after this level.

Each of the AMs has its own set of policies and tries to optimize the performance of its local system. Manager AM2 (see Figure 7.1) manages physical server "1", trying to optimize its performance and

behaviour based on the policies given to it. This includes monitoring the other VMs (VM1 and VM2) in order to help them when they are in need. Because AM2 is running in Domain 0, which is a privileged domain, it can change/resize VMs.

Although we have implemented this system for only three levels of hierarchy, the architecture and concepts used are generalizable to the larger systems such as an entire organization, a data center, etc. Figure 7.2 shows the hierarchy and relationship between AMs in our system.

7.1.2 Policies

In order to define our managed elements, we implemented multiple *ManagedElementInfos* (*MEIs*-See Definition 1) as Java classes. *ApacheMEI*, *VirtualMachineMEI*, *SystemMEI* and *NodeMEI* are defined as the managed elements information. Therefore, an instantiation of these *MEIs* will act as the *MEOs* (e.g. Java objects, Definition 2) in our system. Appendix A shows the metrics and actions available in these *MEIs*.

After defining *MEIs*, we can define policies being used at different levels of the management hierarchy. AM1 and AM3 are running at the virtual machine level and therefore they have *ApacheMEI*. AMs at this level are usually meant to preserve applications SLAs and optimize the virtual machine's performance. In our implementation, AM1 is trying to keep Apache's response time below 500ms as defined in an SLA and has this policy:

OnEvent: ManagementInterval
if ApacheMEI.ResponseTime > 500 **then**
    ApacheMEI.IncreaseMaxClient(25,200);
end if

This policy checks the Apache response time on every *ManagementInterval* and if it's above 500ms, it increases the *MaxClients* property of the Apache web server by 25. The max limit for this property is 200. Therefore, it can not be increased to more than 200 and if this action fails due to this local limitation the message inference algorithm (e.g. Algorithm 5.6) will automatically create a *NOTIFY* message and sends it to the parent manager.

At a higher level, AM2 and AM5 are running at the physical machine level and try to optimize the performance of the physical server by balancing the resources among virtual machines. They have access to *VirtualMachineMEI* and *SystemMEI* and two important policies that are used at this level are:

OnEvent: HelpRequest
if VirtualMachineMEI.MemUtil>85 &
VirtualMachineMEI.CPUUtil>95 **then**
    VirtualMachineMEI.IncreaseMem(50, limit);
end if

This policy checks the memory and CPU utilization of a specific virtual machine upon receiving a *HelpRequest* message and if they

are above certain thresholds (e.g. 85 and 95) it then increases the memory assigned to that virtual machine by 50MB based on available memory in the physical server. This available memory is included in the "limit" variable. "limit" specifies the maximum possible memory that this virtual machine can have (e.g. in this case 500MB) and it can be changed over time. Note that this policy is running on AM2 at Domain 0 which is a privileged domain and therefore AM2 has the authority to change virtual machine's memory. Another policy running at this level is:

OnEvent: Migration
if true **then**
    VMName = SystemMEI.findBestVM()
    SystemMEI.MigrateVMTo(VMName, Migration.NodeName);
end if

This policy shows that upon receiving a *Migration* event it should migrate a virtual machine to the destination specified in the *Migration* event (e.g. *Migration.NodeName*).

This policy says that upon receipt of a *Migration* event at AM2 (Node Controller Level), find the best VM (e.g. least busy), and migrate it to the node specified in the migration event. After a successful migration, it increases the available free memory limit. *SystemMEI* has access to all VMs running in this server and can find the least busy VM and migrate it to another server.

AM4 is one level higher in the hierarchy. It is running at the

cluster controller level and has an overview of all physical machines in that cluster. It has access to *NodeMEI* and *SystemMEI*. A policy that is running at this level is:

OnEvent: HelpRequest
if NodeMEI.MemoryUtil > 50 **then**
BestNode = SystemMEI.findBestNode()
SystemMEI.sendMigrationNotifyMsg(NodeMEI.Name, BestNode)
end if

This policy says that upon receipt of a *HelpRequest* event by AM4, if the server asking for help has a memory utilization of more than 50% then find the best node in the cluster (e.g. the least busy) and generate a *Migration* event and send it to the needy AM in a NOTIFY message. It basically finds the least busy node and notifies the needy child to migrate one of its virtual machines to that node.

We use a greedy approach (e.g. least busy) both for finding the best node and the best virtual machine for migration. We ran three different experiments and measured the Apache response time SLA violations in each scenario.

7.1.3 Scenario 1: No Collaboration

In the first scenario we disabled all communications between managers. In this case, the local managers tried to optimize the system only based on policies that they had and with no further communi-

cation with another manager. Figure 7.3 shows the response time of the Apache web server in this case.

In this case, when the load increases the local manager tries to adjust the web server by allocating more resources. For example at points A, B, C and D in Figure 7.3 an SLA violation was detected by the manager. In response to the SLA violation at points A, B, C and D and based on the policies explained before, the autonomic manager (AM1) increased the *MaxClients* property of the Apache server that it was managing by 25. After point D it also detects an SLA violation, but cannot increase *MaxClients* since it has already reached the maximum value for the *MaxClients* property (i.e., 200).

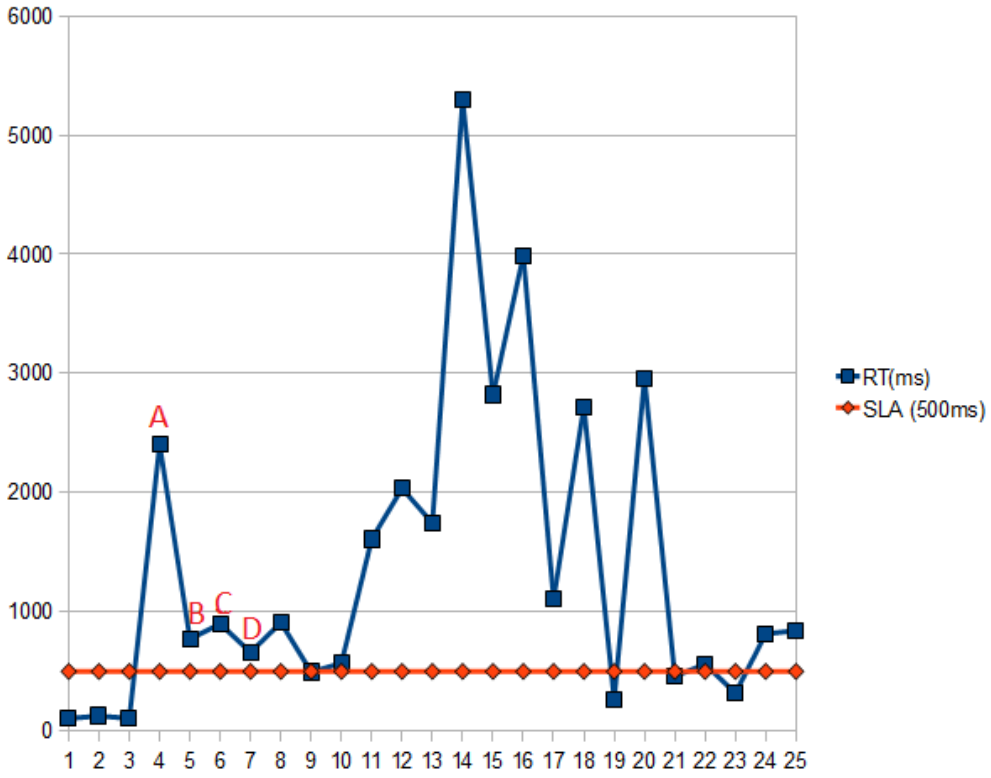

Figure 7.3: Apache response time with no manager collaborations

As a result, the system will face more SLA violations and the response time will get worse (see Figure 7.3). Thus, the load is more than what this system can handle alone. This also causes a long term violation of the SLA (e.g. Apache response time above 500 ms) which could mean more penalties for the service provider.

We calculate two performance measures in this case: the total time that the system could not meet the SLA (T) and the percentage of time that the system spent in a "violation" (V). For these experiments each time interval was 1 second and the x-axis in the graph shows the time interval. Therefore, the results for Scenario 1 are:

T1 = 18 seconds
S1 = Total Experiment Time = 25 seconds
V1 = T1/S1 = 0.72 = 72%

7.1.4 Scenario 2: One Level Collaboration

In the second scenario, we consider the situation when the local manager can request help. When the local manager can no longer make adjustments to the system, it requests help from the higher level manager. This is specified in the policies of AM1 and AM2, as mentioned in the previous section.

The VMs starts with 400MB of memory already allocated to them. The current limit for increasing memory is set to a default value (e.g. limit = 500MB, meaning the max memory this VM can have is 500MB) but it can change over time based on the changes in

the system. We will see an example of this in Scenario 3. Figure 7.4 shows the Apache response time in this case.

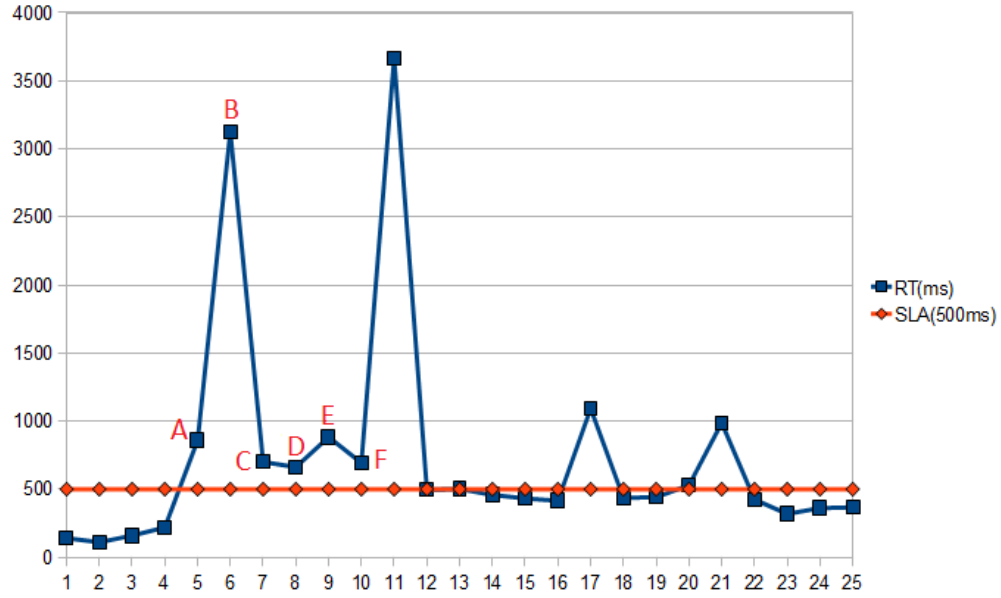

Figure 7.4: Apache response time with one level of collaboration

As in the previous scenario, the local manager (AM1) tries to adjust the web server to handle the increasing load at points A, B, C and D. Eventually, there are no more local adjustments possible (after D) and so the local manager does nothing. In this case, however, when the next SLA violation happens (point E), AM1 generates a *HelpRequest* NOTIFY message and sends it to AM2. In response, AM2 allocates more memory to VM1 (according to its own policies). At this point, the response time starts decreasing, but since the load is still high, AM1 detects another SLA violation at point F and asks for help again, and AM2 allocates 50 more megabytes of memory to VM1, which will reach the maximum allowed memory for the VM (since limit is 500MB).

After the adjustment of memory at point F, there is a sharp spike in the response time as the VM is adjusted to accommodate the increase in memory allocated to it. Moreover, the load is also increasing as well because of the ramp up period configured in JMeter. Once this is completed, the response time decreases. There are still subsequent instances where there are occurrences of heavy load and occasional SLA violations still happen. In these cases, AM1 still sends the help request to AM2, but since AM2 has allocated all available memory to VM1 (as per its policy), it cannot do more and simply ignores these requests. To solve this problem, we add another level of management to the system which is explained in scenario 3. Based on the output for this scenario, we calculated the same measures of performance:

T2 = 10.5 seconds
S2 = 25 seconds
V2 = T2/S2 = 0.42 = 42%

As is evident in the graph (Figure 7.4), the time that the system spends in "violation" of the SLA is much less.

7.1.5 Scenario 3: Two Level Collaboration

In the third and final scenario, we use another level of management to help reduce the occasional SLA violations that happened in Scenario 2. Figure 7.5 shows the Apache response time in this case.

Like the previous scenarios, the local manager (AM1) tries to adjust the web server at points A, B, C and D. At points E and F, AM2

assigns 50 more megabytes to VM1 to solve the stress. At point G there is another SLA violation. At this point, AM1 asks for help from AM2 but since AM2 already assigned all the available memory as per its policy, it cannot provide more help and automatically creates a help request NOTIFY message which it sends to its parent (AM4; see Figure 7.1 and Figure 7.2).

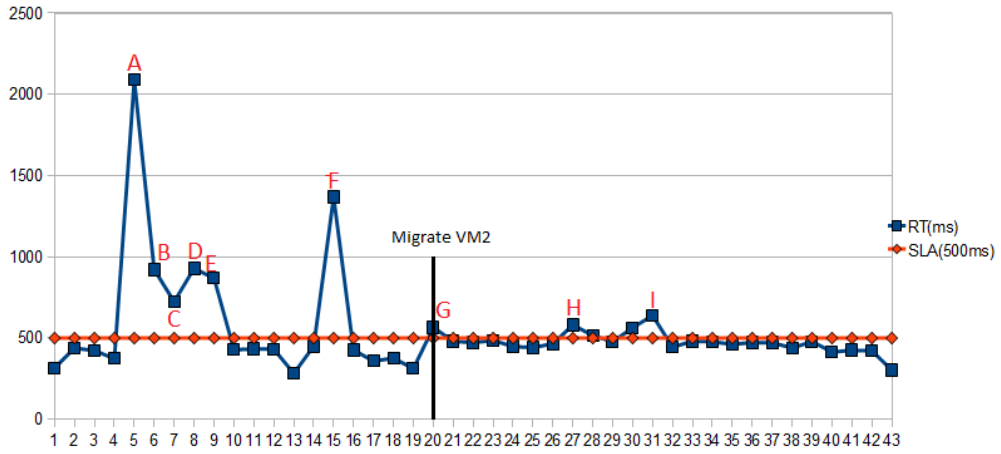

Figure 7.5: Apache response time with two levels of collaboration

AM4, running at the cluster controller level, has a global view of all physical servers and finds the least busy server. It then tells the AM2 to migrate one of the VMs to that server based on the policies explained before. AM2 can then use the host name passed to it in the NOTIFY message to migrate one of its VMs and reduce some load in the server again based on its policy explained in the previous section.

When AM2 receives the NOTIFY message on migration, it chooses a VM to be migrated to the new server. In our implementation, we adopted a greedy approach in both finding the best physical node and finding the best VM for migration. We choose the least busy (based on memory utilization) VM to be migrated. After this VM

is migrated, then there will be more memory available for the busiest VMs. In this case, AM2 migrates VM2 (it had lower memory utilization) to Server2 and removes it from the list of its children. Note that VM1 is the virtual machine in stress situation but VM2 which was less busy (e.g. had lower memory utilization) was chosen for migration. We used the live migration capability in KVM to migrate this VM. Therefore, neither of the virtual machines stopped working during migration.

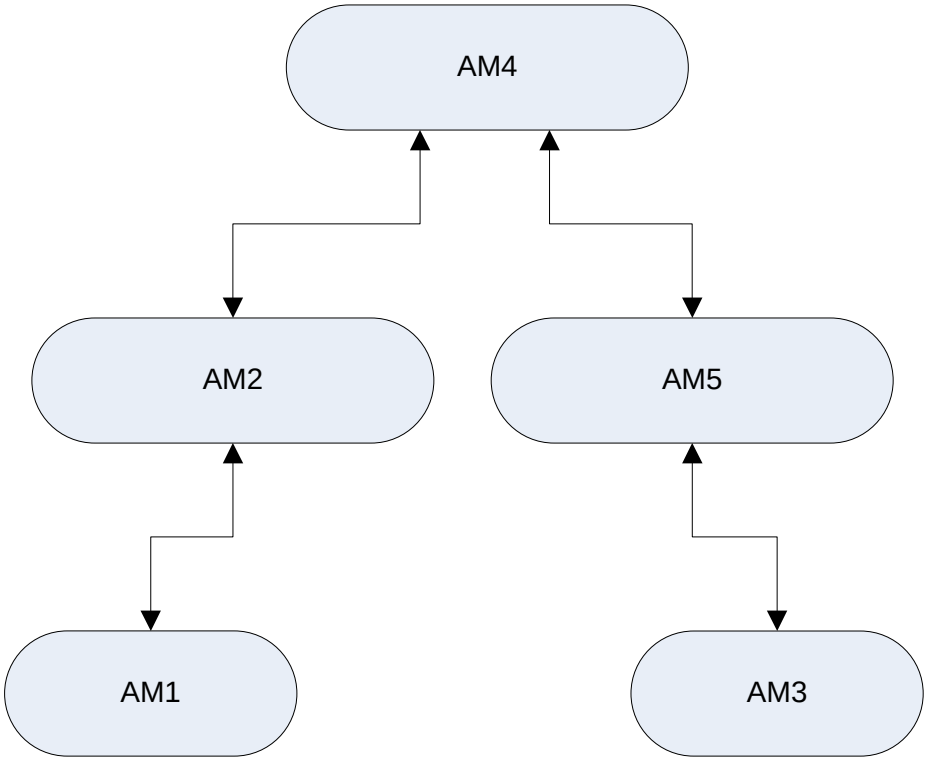

Figure 7.6: Managers hierarchy after migration of VM2 to Server 2

Figure 7.6 shows the hierarchy of AMs after this dynamic change in the VMs structure. In this case, after migration, there is more memory available at the AM2 level and the memory limit is increased. Therefore, at point H (Figure 7.5) when the load is getting

higher and another SLA violation happens, AM1 asks for help and AM2 responds by adding 50 more megabytes to VM1. The same process happens at point I where AM2 adds another 50MB to VM1 (reaching the total of 600MB) and after that the response time stays below the SLA threshold although the load is still very high. The calculation of our measures for this scenario is as follows:

T3 = 10.5 seconds
S3 = 43 seconds
V3 = T3/S3 = 0.24 = 24%

In this case, even with the migration of one of the VMs, the percentage of time in a "violated" state is much less than in Scenario 2.

7.1.6 Discussion

Table 7.3 summarizes the percentage in a "violated" state for the three scenarios. Not surprisingly, having more AMs making changes to the system and components decreased the impact of violations. Most importantly, this happened automatically without administrator intervention and without adding any new hardware which means improvement in the current system efficiency.

The results show that there is definitely an advantage when AMs can collaborate. A single autonomic manager cannot solve all performance problems just by itself because it has only a local view of the system with some limited authority to change things. Thus, the current infrastructure can be used more efficiently and provide bet-

Table 7.3: Results of three scenarios

Scenario	SLA Violation(%)
1: No collaboration between AMs	72
2: One-Level collaboration in the hierarchy	42
3: Two-levels collaboration in the hierarchy	24

ter services with less chance of violating SLAs without adding new computational resources.

7.2 Case Study: High Frequency Trading

CTS is a private company that I had the opportunity to spend an internship working with them on the application of some of these ideas. They are interested in autonomic management of their infrastructure and the research area of this thesis is highly relevant to their management requirements.

CTS develops automated trading technology for financial firms. Their client base includes hedge funds, brokers, banks and professional traders. Their solutions enable the creation of trading algorithms, co-located global deployment, custom connectivity and the automation of entire strategies or portfolios across all asset classes. They work closely with clients to deliver cutting-edge, competitive and cost effective proprietary trading solutions. The company provides a high frequency trading framework for building and running trading algorithms that can perform in real-time. This infrastructure will be referred to as Cloud Trader (CT). Cloud Trader is an automated trading solution that enables development, testing and global deployment of proprietary algorithmic trading strategies.

Clients can develop and test their algorithms using this framework and then launch, monitor and control the algorithm's behaviour. CT is composed of several parts which are running in a distributed manner across the company's private cloud infrastructure. There are thousands of different trading algorithms running at the same time on different virtual machines.

The automation framework has a Complex Event Processing(CEP) system which is designed to analyze massive amounts of market data in real-time, providing rapid identification and response to trading opportunities. The multi-threaded engine separates tasks by different threads on different cores, resulting in a highly scalable system that can support thousands of algorithms running simultaneously. Parallelization and load balancing further ensure consistent high performance of all running algorithms.

Every client has at least one virtual machine which hosts their trading algorithms and can be expanded to more VMs if there is a demand for it. Due to the vast number of framework elements and fast changes in the environment there needs to be a way of managing the whole infrastructure in near real-time without human intervention. The trading orders are being performed in the matter of milliseconds and any change in the infrastructure can have a great impact on a client's revenue as they might lose a lot of money.

Since Cloud Trader is composed of several parts that are being deployed separately and each has a certain role in the system, hav-

ing multiple managers each responsible for managing a particular element seems to be a good solution for managing the whole infrastructure. The infrastructure is very similar to the IaaS clouds since within the Cloud Trader architecture there are multiple layers that require management: there is the algorithm layer which represents trading algorithms that are running inside a virtual machine, there is the virtual machine layer, there is the host machine layer and so on.

Given this, it seemed that an approach similar to our proposed hierarchical system would be a good strategy towards management of this infrastructure. AMs could be installed with the managed elements in different parts of the system and would be started whenever their corresponding managed element was started. For example, when a new virtual machine gets installed, its manager would also be installed with it, or when a new application is installed its manager would be installed with it. Therefore, upon starting up a virtual machine the AM inside that virtual machine is automatically started and will contact the registry to find the right position in the hierarchy.

7.2.1 Management Architecture

The physical layout of the experimental system is illustrated in Figure 7.7. The virtual machine is running on a Windows server 2008 with hyper-v virtualization and has a Windows 7 installed on it. One of the company applications called "Sliver" is deployed as a windows service inside the virtual machine. Sliver is responsible for facilitating the communication between trading algorithms running inside the virtual machine with the outside world. Both Sliver and the VM

have their own agent which is installed with them.

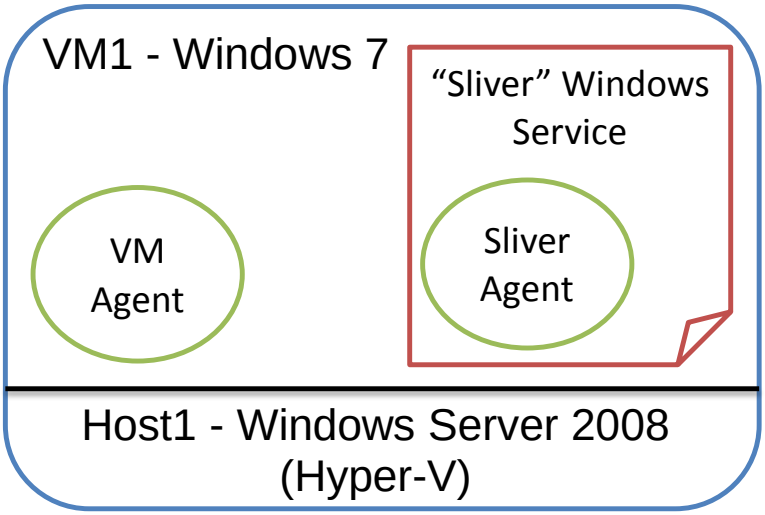

Figure 7.7: Case Study Physical Layout

We implemented this system for two levels of the hierarchy (shown in Figure 7.8). In the first level, there is an autonomic manager for Sliver that monitors the Sliver behaviour. At the higher level, there is an autonomic manager for a virtual machine which monitors the health status of a typical VM. This manager can monitor metrics like CPU utilization, memory utilization, service status, etc. and enforce related policies.

Therefore there are two management groups (see Chapter 6) at two different levels and Table 7.4 shows the management groups in this system. Table 7.5 show the attributes of each management group including the *MEIs*.

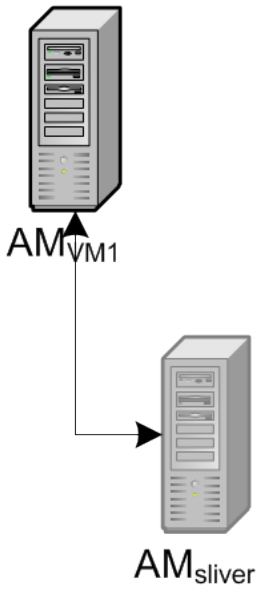

Figure 7.8: Management's Hierarchy - two levels

Table 7.4: CTS Management Groups

Level	MG Name
0	VirtualMachineMG
1	SliverMG

Table 7.5: CTS MGAttributes

MG Name	MEI(s)	Technology Script Name
VirtualMachineMG	VirtualMachineMEI	Windows.wsf
SliverMG	SliverMEI	Sliver.wsf

7.2.2 Implementation

We implemented a central policy repository and each manager can retrieve related policies at start-up time. We also implemented the central registry where managers are registered upon start-up and tested the start-up (Algorithm 5.1) and termination detection (Algorithm 5.5) algorithms.

We implemented these ideas within the CTS infrastructure. The company also wanted the management framework to be consistent

and integrated with their internal software and therefore this was considered during the design of the management framework. We used the C# programming language for the autonomic managers and used Microsoft's BizTalk [24] rule engine for the policy evaluation. All policies are defined using the BizTalk rule composer and stored in a Microsoft SQL server as the central repository for all policies. We also used Windows server 2008 and Hyper-V technology to host Windows 7 virtual machines. All *ManagedElementInfos(MEIs)* are defined as C# classes.

In order to integrate this management system with the current CTS infrastructure and deploy them easily across different parts of the CTS infrastructure, we split each manager into two different parts: 1) The monitoring and action execution part is implemented in an "Agent" (the sensor/actuator part of the manager); 2) The policy processing and decision making part is implemented and run in a different process. By separating the sensor/actuator part of the autonomic manager from the decision making part (policy evaluation) we introduce a way for the agents to be installed and run in a loosely-coupled manner which does not affect the functionality of the rest of system. If the decision making part needs to get updated or changed it will not affect the sensor/actuator elements in the core part of the operational system. It also introduces the possibility of detecting terminated elements. For example, if the virtual machine shuts down, then the agent inside the VM will also shut down but the manager is still alive and can detect this situation. Similarly, if the virtual machine turns back on, the agent inside the VM will start working again and the manager can now detect that and en-

force relevant policies. This facilitates the installation and removal of managers in the hierarchy.

For example, a virtual machine autonomic manager would have a virtual machine agent which will be installed and run inside of the virtual machine. This agent is able to monitor different parts of the VM and report them to the decision making part which is running somewhere else in the system (outside of the VM). The agent is also able to perform any action that its decision making component asks it to do. The agents basically act as a sensor/actuator in the system.

Each agent is configured with the AM name and parent name. For example, the AM name for an agent that is monitoring a virtual machine is the virtual machine name and can be obtained automatically when the program starts running, however the parent name is the host machine name in which the virtual machine is running and this should be set as a configuration parameter. Each agent will contact the central registry upon start-up and send a message to its manager.

Each agent gathers information and creates a *ManagedElementObject(MEO)* when a useful event happens. The *MEO* will then be sent to its manager. The manager evaluates all relevant policies against the received *MEO*, updates the output actions and returns the *MEO* to the agent. The agent then inspects the received *MEO* and executes code to satisfy the result actions.

The overall data flow for a sample host machine agent is illustrated in Figure 7.9:

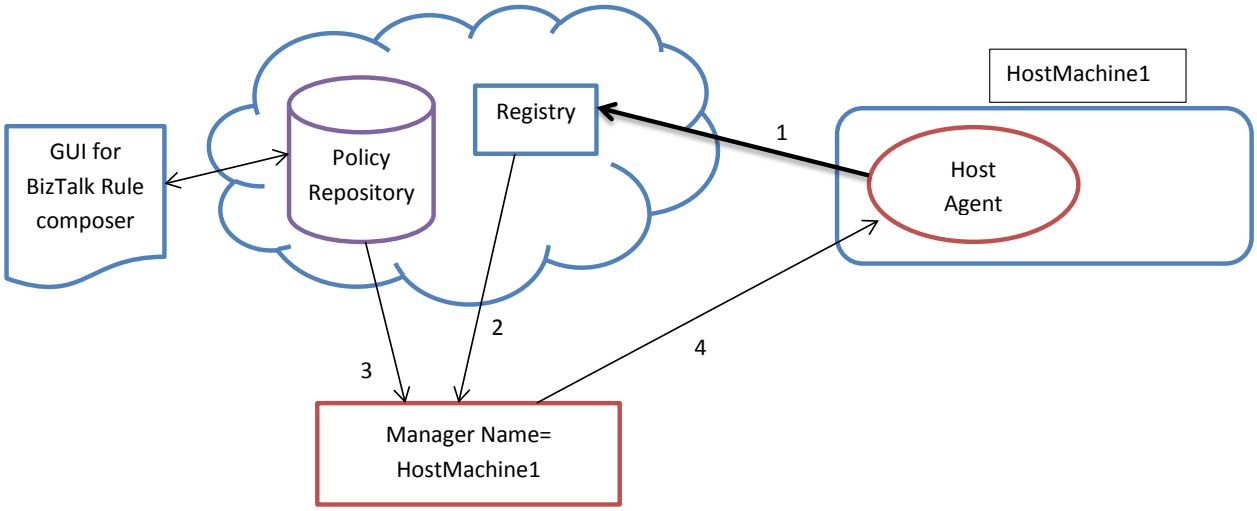

Figure 7.9: Data flow for a host machine agent

1. The host agent gets installed on the host machine through a Microsoft MSI installer as a Windows service. It then register itself in the central registry to contact its manager.

2. Registry will register the name and start the relevant manager (if not running).

3. The Manager loads the policies related to this agent (e.g. if it is a host agent, it would load host related policies) and evaluate them. This includes running the configuration policies for the first time.

4. The Manager then sends the result of that policy evaluation back to the agent for enforcement, which includes the configuration parameters or actions that have to be performed on that machine. After a successful agent configuration, the agent starts monitoring the host based on the thresholds and configuration parameters and will notify its manager with the new *MEOs* from time to time.

The central registry is also implemented in C# and uses some of the libraries developed by the company. It receives incoming messages from agents and passes them on to the right manager. If the relevant manager is dead, it will start it and pass the incoming message. The manager will then add that agent to the list of its managed elements and enforce the policies by sending a response message back to the agent. When the agent contacts its manager for the first time, the manager will enforce the configuration policies.

The system administrator can view, edit and deploy different policies at run time through the Biztalk rule composer GUI which will then get updated in the policy repository and be used by the managers.

7.2.3 Policies

There are two types of policies that are used in the system: configuration policies and operational policies. An example of a configuration policy used at the virtual machine level is the following:

OnEvent: VMManagedObjectReceived
if VirtualMachineMEI.getConfMode()= true **then**
    VirtualMachineMEI.setCPUUtilizationThreshold(85);
    VirtualMachineMEI.setRefreshInterval(2000);
end if

A VM agent sends the *VMManagedObjects* to the manager to report the status of different metrics. Based on this policy, upon receiving a new *VMManagedObject*, if the agent is in configuration

mode, it will then set the CPU utilization threshold to 85% and set the refresh interval for monitoring the CPU utilization to 2000ms. The *VMManagedObject* will then be sent back to the agent for enforcement. After a successful configuration, the agent will check the CPU utilization every 2000ms and will report it to the manager only if it goes above 85%.

An example of an operational policy used at the virtual machine level is:

OnEvent: AMTermination
if $VirtualMachineMEI.getServiceStopped() =$ "*Sliver*" **then**
    VirtualMachineMEI.startService("Sliver");
    VirtualMachineMEI.sendEmailTo("x@company.com");
end if

If the Sliver Windows service dies (as explained before, Sliver is one of the applications of the company), its agent gets terminated and its manager will also stop working because the TCP connection of the registry and the agent will be terminated and the registry will notify the manager about this termination. At this point, the higher level manager (AM_vm1) will detect removal of its child AM (see algorithm 5.5) and raise an *AMTermination* event. In this policy, the manager checks to see if the name of the service stopped is equal to "Sliver". If the service stopped is in fact the Sliver service it will then start that service by telling the virtual machine agent which service to start and send an email to the responsible person to report this failure. After a successful start of the Sliver service, its

agent starts working again and therefore its manager will get added to the management hierarchy automatically (see Algorithm 5.1).

These policies are defined by using Microsoft Biztalk rule composer graphical user interface (GUI) and stored in the SQL server database. The system administrator can change and redeploy these policies at any time on the fly. All traffic between agents and managers are transmitted over TCP and a secure administrator network.

7.2.4 Lessons Learned

During the four months internship period, we were able to implement and test managers at only two levels of the hierarchy, but have the related machinery implemented, the registry, the policy repository, etc. We were also only able to do limited testing and evaluation. However, we can make some observations:

- While it did take some time to implement the underlying support mechanisms, the creation of AMs (the agent (sensor/actuator) part and decision making part) has gone well and is straightforward. The decision making part is very similar across the AMs and the real dependencies are in the metrics to collect and actions to take for different management elements, i.e., the specific sensors and actuators.

- Separating the sensor/actuator and decision making parts of the AMs has worked well. Once decisions are made on what data to collect and what actions can be taken, the sensor/actuator part can be implemented and left. Different behaviours can be

accommodated through the policies specified.

- The start-up and termination detection algorithms work well in the hierarchical approach and these algorithms run fast enough in the context they were used; something very important to the company given their domain.
- The central policy repository makes it easier for administrator to manage different policies from a single point and modify these policies on the fly based on new requirements and without affecting the running autonomic managers.
- The naming registration and central registry worked well and the communication between agents and managers was facilitated by this registry.

The main points tested in this implementation is the exploration of central policy repository as well as testing the start-up, processing, policy evaluation and termination detection algorithms explained before.

7.3 Summary and Discussion

As part of the testing of the proposed management model, we implemented a prototype in a small cloud environment and evaluated hierarchical organization of managers, automatic message inference mechanism and collaboration of multiple managers using the communication protocol developed in this work. We also explained what policies look like at different levels of this hierarchy and how one can enforce policies at different authoritative locations. The important point is that all of this is happening automatically with no human

intervention.

We also explained a practical case study and applied our ideas in developing a hierarchical management system for a private company. We tested central repository, registry techniques and start-up and termination detection algorithms.

The communication protocol seems to be general enough in the context that we tested it, which can cover various types of messages with detailed information to be sent from one manager to another and it can also be used in other types of organizations (e.g. it is not only limited to a hierarchical organization).

Automatic message inference algorithm helps to automate the communication process between managers which leads to faster reactions to dynamic changes in the environment.

Central policy repository and registry with start-up algorithm helps to automate the process of policy distribution between managers, facilitate the collaboration process and makes future policy updates easier. It also helps building a dynamic hierarchy that can restructure on the fly.

Overall, the autonomic management model proposed in this thesis seems to be a good approach for monitoring and management of large computing environments where there are multiple managers involved. This model helps building more automated clouds that use their resources more efficiently while meeting their users' require-

ments.

Chapter 8

Conclusion

8.1 Summary

We explored the use of multiple autonomic managers in a computing environment to facilitate the autonomic management of that environment. We addressed the problem of how different autonomic managers should be organized in a large computing environment such as an Infrastructure-as-a-Service (IaaS) cloud, how they should interact with each other to achieve the goals of the system and when this communication should happen [29, 31, 30]. We also explored how different autonomic managers should be deployed across the infrastructure and how we could automate this deployment process. However, a particular focus of this research was on IaaS clouds as a good infrastructure to apply our ideas.

More specifically, we focused on the following problems:

- How should "multiple" autonomic managers collaborate with each other in a large computing environment to achieve global goals?
- How to automate the collaboration of managers in the system?

In order to deal with a dynamic environment where applications can start and stop and where virtual machines may come an go, there is a need to ensure that managers can communicate and collaborate. How can communication between managers be defined in a changing environment as managers come and go? How is the communication structured and what is exchanged?

- What is a scalable approach for the deployment of autonomic managers? What is a good strategy for deploying these managers so that it requires minimal manual administrative efforts?

- How can autonomic managers detect the addition or removal of different elements and automatically restructure the hierarchy of managers without human intervention? How does the management hierarchy restructure on the fly to reflect these changes?

- How to automate the manager configuration and minimize the administrative costs to setup autonomic managers? Each autonomic manager needs to be configured before or upon start-up. However, in a large system configuring all managers one by one can become a challenging and error prone job for administrators. How can this process be automated to help administrators and reduce the costs associated with it?

We proposed a hierarchical approach towards management of such systems and developed a communication protocol between autonomic managers. We used policies as a means of describing operational behaviour and SLA definitions and showed how some of the communication messages can be inferred from these policies automatically based on the demand. We focused on some practi-

cal challenges in the management and use of multiple autonomic managers and explained how multiple policy-based autonomic managers organized in a hierarchical fashion can monitor and manage an Infrastructure-as-a-Service type of cloud. We developed several algorithms which describe the behaviour of a particular autonomic manager and addressed issues of automatic deployment, termination detection and configuration of managers and proposed a novel solution that is easy to maintain.

We tried to keep the number of messages that used for communication between two managers limited in order to keep message overhead reduced to the extent that was possible. The measurement of the actual overhead and the scalability testing of the management system is part of the future work since testing the scalability of this approach requires a lot of physical resources or should be considered in a simulated environment which is beyond the scope of this thesis.

8.2 Main Contributions

The main contributions of this work and the novel ideas are as follows:

- There has been generally a little work in the area of multiple autonomic managers and how to handle dynamic changes. Therefore, this work is to somewhat unique in this area.
- Cluster management typically has a focus on the cluster as a whole often ignoring management of individual elements, such as nodes. Our hierarchical approach in this thesis encompasses

a focus on local and intermediate managers as well as including global cluster level managers which makes it unique in addressing this problem.

- The design of a hierarchical autonomic management model for large computing environments with formal definition of different elements in that model (Chapter 4).

- The design of a communication protocol between autonomic managers that facilitates their collaboration in achieving global goals (Section 5.2). Some of these communication messages can be inferred from policies and therefore can help with automating the collaboration between managers.

- Introduction of multiple algorithms that define the behaviour of a specific autonomic manager and its relationship with other managers in that management model. These algorithms include the start-up, processing, termination detection and communication message inference from policies (Chapter 5).

- Design of a deployment system based on the management model proposed to automate the deployment of different autonomic managers across the computing environment with minimum administrative efforts (Chapter 6).

- Creation of multiple algorithms as part of this deployment system such as element discovery, members addition and members removal (Section 6.4 and Section 6.5). The time complexity of element discovery algorithm is $O(n^2)$ where n is the number of AMs that should be deployed in the whole computing environment (e.g. number of nodes in the management tree). The time

complexity of members addition algorithm is O(log(n)) and the members removal is O(n) in the worst case.

We also evaluated these ideas in two different experimental settings. In one case, we implemented this approach in a small private cloud and measured the potential advantages of a hierarchical approach. We also implemented some of our ideas and algorithms in a real world setting involving a high frequency trading cloud infrastructure.

8.3 Future Work

Cooperating autonomic managers for managing a cloud infrastructure seems to offer some promises. The hierarchical organization of managers has advantages and seems to be a good approach in the application domain in which we used it. However, there are other means of organizing managers that need to be investigated (e.g. peer-to-peer).

Specific items for future work include:

- Considering other types of organizations for autonomic managers (e.g. peer-to-peer approach) and compare it with the current hierarchical structure.
- There will be more autonomic managers deployed in the system as the number of levels increase in the management hierarchy. As part of the future work, it would be interesting to see what is the overhead of this management model (e.g. in terms of

network traffic due to communication messages) where there are more levels of hierarchy involved.

- It would also be interesting to see what policies will look like in higher level managers when the levels of the management hierarchy increase.

- There has been a lot of research about policy decomposition. Are those methods applicable to a management model like this? or does the hierarchical organization of managers in this model help to facilitate the decomposition process?

Further work on this approach can lead to more automated management of cloud environments enabling more efficient use of the cloud infrastructure and as well as meeting SLA requirements while using fewer resources.

Bibliography

[1] Apache JMeter Load Generator. `http://jakarta.apache.org/jmeter/`. [Online; accessed Jan 2012].

[2] Virtuemart Online Shop. `http://virtuemart.net/`. [Online; accessed Jan 2012].

[3] Dakshi Agrawal, Seraphin Calo, Kang-Won Lee, Jorge Lobo, and Dinesh Verma. *Policy technologies for self-managing systems*. IBM Press, 2008.

[4] Marco Aldinucci, M. Danelutto, and P. Kilpatrick. Towards hierarchical management of autonomic components: a case study. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 3–10. IEEE, 2009.

[5] Anton Beloglazov and Rajkumar Buyya. Managing Overloaded Hosts for Dynamic Consolidation of Virtual Machines in Cloud Data Centers under Quality of Service Constraints. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1366–1379, July 2013.

[6] Raouf Boutaba and Issam Aib. Policy-based Management: A Historical Perspective. *Journal of Network and Systems Management*, 15(4):447–480, November 2007.

[7] D.W. Chadwick, A. Basden, and J.a. Cunningham. Automated Decomposition of Access Control Policies. In *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, pages 3–13. IEEE, 2005.

[8] Sivadon Chaisiri and Dusit Niyato. Optimal virtual machine placement across multiple cloud providers. In *2009 IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 103–110. IEEE, December 2009.

[9] Antonio Corradi, Mario Fanelli, and Luca Foschini. VM consolidation: A real case based on OpenStack Cloud. *Future Generation Computer Systems*, June 2012.

[10] R. Craven, J. Lobo, and E. Lupu. Policy refinement: decomposition and operationalization for dynamic domains. In *Network and Service Management (CNSM)*, 2011.

[11] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Policies for Distributed Systems and Networks*, pages 18–38. Springer Berlin Heidelberg, 2001.

[12] Rajarshi Das, J.O. Kephart, and C. Lefurgy. Autonomic multi-agent management of power and performance in data centers. In *Proceedings of the 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AA- MAS 2008)- Industry and Applications Track*, pages 107–114. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[13] Yurdaer Doganata, Keith Grueneberg, John Karat, and Nirmal Mukhi. Authoring and Deploying Business Policies Dynamically

for Compliance Monitoring. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 161–164. Ieee, June 2011.

[14] Jeroen Famaey, Steven Latrea, John Strassner, and Filip De Turck. A hierarchical approach to autonomic network management. In *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*, pages 225–232. Ieee, 2010.

[15] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring Alternative Approaches to Implement an Elasticity Policy. In *IEEE 4th International Conference on Cloud Computing*, pages 716–723. Ieee, July 2011.

[16] Weili Han and Chang Lei. A survey on policy languages in network and security management. *Computer Networks*, 56(1):477–489, January 2012.

[17] Markus C. Huebscher and Julie a. McCann. A survey of autonomic computingdegrees, models, and applications. *ACM Computing Surveys*, 40(3):1–28, August 2008.

[18] J. O. Kephart, H Chan, R Das, D W Levine, G Tesauro, and F R An C Lefurgy. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *in IEEE Intl. Conf. on Autonomic Computing*, pages 145–154, 2006.

[19] J.O. Kephart. Research challenges of autonomic computing. In *27th International Conference on Software Engineering (ICSE)*, pages 15–22. IEEE, 2005.

[20] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[21] Avi Kivity, Y. Kamay, D. Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[22] Sangmin Lee and Rina Panigrahy. Validating heuristics for virtual machines consolidation. *Microsoft Research Technical Report MSR-TR-2011-9*, 2011.

[23] Qiang Li, Qin-fen Hao, Li-min Xiao, and Zhou-Jun Li. An Integrated Approach to Automatic Management of Virtualized Resources in Cloud Environments. *The Computer Journal*, 54(6):905–919, November 2010.

[24] B. Loesgen, J. Charles Young, J. Eliasen, S. Colestock, A. Kumar, and J. Flanders. *Microsoft BizTalk Server 2010: Unleashed*. Unleashed Series. Sams, 2011.

[25] R. Makhloufi and G. Doyen. SAAM: A self-adaptive aggregation mechanism for autonomous management systems. In *IEEE Network Operations and Management Symposium*, pages 667–670, 2012.

[26] Rafik Makhloufi and Guillaume Doyen. Towards self-adaptive management frameworks: the case of aggregated information monitoring. In *Network and Service Management (CNSM)*, 2011.

[27] Patrick Martin, Andrew Brown, Wendy Powley, and Jose Luis Vazquez-Poletti. Autonomic management of elastic services in the cloud. In *2011 IEEE Symposium on Computers and Communications (ISCC)*, pages 135–140. IEEE, June 2011.

[28] Nader Mbarek, M.A. Chalouf, and Francine Krief. Towards global service level guarantee within autonomic computing systems. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium On*, pages 446–453. Ieee, May 2011.

[29] Omid Mola and M.A. Bauer. Collaborative policy-based autonomic management: In a hierarchical model. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–5, 2011.

[30] Omid Mola and Mike Bauer. Policy-Based Autonomic Collaboration for Cloud Management. In *The Seventh International Multi-Conference on Computing in the Global Information Technology (ICCGI)*, number c, pages 288–293, Venice, Italy, 2012.

[31] Omid Mola and Michael A. Bauer. Towards Cloud Management by Autonomic Manager Collaboration. *Int'l J. of Communications, Network and System Sciences*, 04(12):790–802, 2011.

[32] Tridib Mukherjee, Ayan Banerjee, Georgios Varsamopoulos, and Sandeep K.S. Gupta. Model-driven coordinated management of data centers. *Computer Networks*, 54(16):2869–2886, November 2010.

[33] Richard Murch. *Autonomic Computing*. IBM Press, 2004.

[34] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, 2009.

[35] A. Ouda, Hanan Lutfiyya, and Mike Bauer. Automatic Policy Mapping to Management System Configurations. In *2010 IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 87–94. IEEE, 2010.

[36] Manish Parashar and Salim Hariri. Autonomic computing: An overview. *Unconventional Programming Paradigms*, 3566:257–269, 2005.

[37] Indrani Paul, Sudhakar Yalamanchili, and Lizy K. John. Performance impact of virtual machine placement in a datacenter. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, pages 424–431. IEEE, December 2012.

[38] Alexander Pokluda, Gaston Keller, and Hanan Lutfiyya. Managing dynamic memory allocations in a cloud through golondrina. In *4th International DMTF Academic Alliance Workshop on Systems and Virtualization Management (SVM)*, pages 7–14. Ieee, October 2010.

[39] Mustafizur Rahman, Rajiv Ranjan, and Rajkumar Buyya. A taxonomy and survey on autonomic management of applications in grid computing environments. *Concurrency and Decomposition: Practice and Experience*, (May):1990–2019, 2011.

[40] Mazeiar Salehie and L. Tahvildari. A policy-based decision making approach for orchestrating autonomic elements. In *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pages 173–181. IEEE, 2005.

[41] Mazeiar Salehie and L Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2), 2009.

[42] Alberto Schaeffer-Filho, Emil Lupu, Naranker Dulay, Sye Loong Keoh, Kevin Twidle, Morris Sloman, Steven Heeps, Stephen Strowes, and Joe Sventek. Towards Supporting Interactions between Self-Managed Cells. In *First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, number Saso, pages 224–236. IEEE, July 2007.

[43] Alberto Schaeffer-Filho, Emil Lupu, and Morris Sloman. Realising management and composition of self-managed cells in pervasive healthcare. In *Pervasive Computing Technologies for Healthcare, 2009. PervasiveHealth 2009. 3rd International Conference on*, pages 1–8. IEEE, 2009.

[44] Lars Christoph Schmelz, Mehdi Amirijoo, Andreas Eisenblaetter, Remco Litjens, Michaela Neuland, and John Turk. A Coordination Framework for Self-Organisation in LTE Networks. In *12th Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium On*, pages 193–200, 2011.

[45] Vivek Shrivastava, Petros Zerfos, Kang-won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers. In *2011 Proceedings IEEE INFOCOM*, pages 66–70. IEEE, April 2011.

[46] Bradley Simmons, Hamoun Ghanbari, Sotirios Liaskos, Marin Litoiu, and Gabriel Iszlai. Hierarchical Self-Optimization of SaaS Applications in Clouds. In *Software Engineering for Self-*

Adaptive Systems II, pages 354–375. Springer Berlin Heidelberg, 2013.

[47] Bradley Simmons and Hanan Lutfiyya. Achieving High-Level Directives Using Strategy-Trees. In *Modelling Autonomic Communications Environments*, volume 5844, pages 44–57. Springer Berlin Heidelberg, 2009.

[48] M.A. Soares and E.R.M. Madeira. A multi-agent architecture for autonomic management of virtual networks. In *IEEE Network Operations and Management Symposium*, pages 1183–1186, 2012.

[49] Malgorzata Steinder and Ian Whalley. Coordinated management of power usage and runtime performance. In *IEEE Network Operations and Management Symposium(NOMS)*, pages 387–394. IEEE, 2008.

[50] Gerald Tesauro, D.M. Chess, W.E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, J.O. Kephart, and S.R. White. A multi-agent systems approach to autonomic computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 464–471. IEEE Computer Society, 2004.

[51] D. Tuncer, M. Charalambides, G. Pavlou, and N. Wang. DACoRM: A coordinated, decentralized and adaptive network resource management scheme. In *IEEE Network Operations and Management Symposium*, pages 417–425. IEEE, April 2012.

[52] Kevin Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder2: A Policy System for Autonomous Pervasive Environ-

ments. *2009 Fifth International Conference on Autonomic and Autonomous Systems*, pages 330–335, 2009.

[53] Rahul Urgaonkar, Ulas C. Kozat, Ken Igarashi, and Michael J. Neely. Dynamic resource allocation and power management in virtualized data centers. *2010 IEEE Network Operations and Management Symposium - NOMS 2010*, (Vm):479–486, 2010.

[54] Meng Wang, Xiaoqiao Meng, and Li Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. *Proceedings of IEEE INFOCOM*, (L):71–75, 2011.

[55] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, December 2009.

[56] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, April 2010.

[57] Xiangliang Zhang and Z.Y. Shae. Virtual machine migration in an over-committed cloud. *IEEE Network Operations and Management Symposium*, (Vm):196–203, 2012.

[58] Xiaoyun Zhu, Donald Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Robert Gardner, Tom Christian, and Ludmila Cherkasova. 1000 Islands: an Integrated Approach To Resource Management for Virtualized Data Centers. *Cluster Computing*, 12(1):45–57, November 2008.

Appendix A

Managed Element Infos

Listing A.1: LocalMEI.java

```
package net.ponder2.managedobject;

/**
 * imports removed to save space
 */
public abstract class LocalMEI extends P2Object{
private static Logger logger =
        Logger.getLogger(LocalMEI.class);
private String name;
private String amName;

public LocalManagedObject(String name, String amNameInside) {
        this.name = name;
        this.amName = amNameInside;
}

@Ponder2op("getAMName")
public String getAmName() {
        return amName;
}

public void setAmName(String amName) {
        this.amName = amName;
}

public String getName() {
        return name;
```

```
}

public void setName(String name) {
        this.name = name;
}

protected abstract void refreshMetricValues();

@Override
public P2Object readXml(TaggedElement xml,
                Map<Integer, P2Serializable> read)
                throws Ponder2OperationException,
                Ponder2ArgumentException {
        // TODO Auto-generated method stub
        return null;
}

}
```

Listing A.2: ApacheMEI.java

```
package net.ponder2.managedobject;

/**
 * imports removed to save space
 */
public class ApacheMEI extends LocalMEI {
private static Logger logger =
        Logger.getLogger(ApacheMEI.class);
/**
 * MEI Metrics
 */
public float responseTime;
public float totalAccesses;
public float totalKBytes;
public float cpuLoad;
public float upTime;
public float reqPerSec;
public float bytesPerSec;
public float bytesPerReq;
public float busyWorkers;
public float idleWorkers;
public boolean runningStatus = false;
```

```
@Ponder2op("runningStatus")
public boolean isRunning() {
        return runningStatus;
}

@Ponder2op("responseTime")
public float getResponseTime() {
        return responseTime;
}

/**
 * MEI Attributes
 */
private float maxClients;
private float maxKeepAliveRequests;
private float keepAliveTimeout;
private float minSpareThreads;
private float maxSpareThreads;
private float threadsPerChild;

@Ponder2op("maxClients")
public float getMaxClients() {
        return maxClients;
}

@Ponder2op("maxClients:")
public void setMaxClients(float maxClients) {
this.maxClients = maxClients;
writeProperties();
restartServer();
}

private String apachePath;
private String confFilePath;

/**
 * Constructor
 */
public ApacheMEI(String name, String amName) {
super(name, amName);
apachePath = "/etc/apache2";
```

```
if (apachePath == null) {
        throw new RuntimeException("No path for Apache set.");
}
confFilePath = apachePath + "/apache2.conf";
readProperties();
}

/**
 * MEI Actions
 */
public void startServer() {
if (Common.executeCommand("/etc/init.d/apache2 start")
        == null)
logger.error("Could not start apache");
else
        logger.info("apache started successfully");
readProperties();
}

public void stopServer() {
if (Common.executeCommand("/etc/init.d/apache2 stop")
        == null)
logger.error("Could not stop apache");
else
        logger.info("apache stoped successfully");
}

@Ponder2op("restart")
public void restartServer() {
if (Common.executeCommand("/etc/init.d/apache2 restart", true)
        == null)
logger.error("could not restart apache");
else
        logger.info("apache restarted successfully");
readProperties();
}

@Ponder2op("increaseMaxClients:max:")
public void increaseMaxClients(float amountToIncrease, float max){
if (this.maxClients + amountToIncrease <= max)
        setMaxClients(this.maxClients + amountToIncrease);
else {
```

```
        String ponderTalkString =
                "root/event/SendHelpReqEvent create";
        String result;
        try {
                String p2xml = P2Compiler.parse(ponderTalkString);
                P2Object value = new XMLParser().execute(
                                SelfManagedCell.RootDomain, p2xml);
        } catch (Exception e) {
                e.printStackTrace();
        }
}
}

/**
 * Private and protected methods (Helper methods)
 * removed to save space
 */

}
```

Listing A.3: VirtualMachineMEI.java

```
package net.ponder2.managedobject;

/**
 * imports removed to save space
 */
public class VirtualMachineMEI extends LocalMEI{
private static Logger logger =
        Logger.getLogger(VirtualMachineMEI.class);
/**
 * MEI Metrics
 */
private float cpuUtilization;
private float memoryUtilization;
private String ip;
private int memoryMB; // In MB

@Ponder2op("getIP")
public String getIp() {
        return ip;
}
```

```
public float getCpuUtilization() {
        return cpuUtilization;
}

@Ponder2op("memoryUtil")
public float getMemoryUtilization() {
        return memoryUtilization;
}

public int getMemoryMB() {
        return memoryMB;
}

@Ponder2op("setIP:")
public void setIp(String ip) {
        this.ip = ip;
}

private static final int CONVERSION_MB_TO_KB = 1024;

/**
 * Constructor
 */
public VirtualMachineMEI(String name, String amName) {
        super(name, amName);
        readProperties();
}

/**
 * MEI Actions
 */
@Ponder2op("setMem:")
public void setMemoryMB(int mbMem) {
int kbMem = toKB(mbMem);
if (Common.executeCommand("virsh -c qemu:///system setmem "
        + this.getName() + " " + kbMem) == null) {
logger.error("Could not set memory to " + kbMem + " KB");
        return;
} else {
logger.info("set memory Successfully to: " + kbMem + " KB");
this.memoryMB = mbMem;
}
```

```
}

@Ponder2op("increaseMem:max:")
public void increaseVMMemory(int amountToIncrease, int max) {
if (this.memoryMB + amountToIncrease <= max) {
setMemoryMB(this.memoryMB + amountToIncrease);
logger.info("Increased␣memory␣to:␣" + (this.memoryMB));
} else {
String ponderTalkString = "root/event/SendHelpReqEvent␣create";
String result;
try {
        String p2xml = P2Compiler.parse(ponderTalkString);
        P2Object value = new XMLParser().execute(
                        SelfManagedCell.RootDomain, p2xml);
} catch (Exception e) {
        e.printStackTrace();
}
}
}
public void startVM() {
if (Common.executeCommand("virsh␣-c␣qemu:///system␣start␣"
        + this.getName()) == null)
logger.error("Could␣not␣start␣VM␣" + this.getName());
else
        logger.info(this.getName() + "␣VM␣started␣successfully");
readProperties();
}

/**
 * Private and protected methods (Helper methods)
 */
private void readProperties() {
BufferedReader domainInfoReader;
String line = null;
try {
domainInfoReader = Common
        .executeCommand("virsh␣-c␣qemu:///system␣dominfo␣"
                        + this.getName());
while ((line = domainInfoReader.readLine()) != null) {
        if (line.startsWith("Used␣memory:")) {
        memoryMB = toMB(Integer.parseInt((line.split("␣")[5])));
}
```

```
}
domainInfoReader.close();
} catch (IOException e) {
        logger.error("Problem␣reading␣domain␣info", e);
} catch (NumberFormatException e) {
        logger.error("Could␣not␣parse", e);
        }
}

private void writeProperties() {
        int kbMem = toKB(memoryMB);
        if (Common.executeCommand("virsh␣-c␣qemu:///system␣setmem␣"
                + this.getName() + "␣" + kbMem) == null)
        logger.error("Could␣not␣set␣memory");
}

@Override
protected void refreshMetricValues() {
try {
RMIReceiveInterface pt = (RMIReceiveInterface) Naming
        .lookup("rmi://" + ip + "/" + this.getAmName());
memoryUtilization =
  pt.executePonderTalk("root/am/system␣memUtil")
                .asFloat();
cpuUtilization = pt.executePonderTalk("root/am/system␣cpuUtil")
                        .asFloat();
} catch (Exception e) {
logger.error("Getting␣MemUtil␣from:␣rmi://" + ip + "/"
                        + this.getAmName());
        e.printStackTrace();
        memoryUtilization = 0;
        cpuUtilization = 0;
}
}

private int toMB(int kbMem) {
        return kbMem / CONVERSION_MB_TO_KB;
}

private int toKB(int mbMem) {
        return mbMem * CONVERSION_MB_TO_KB;
}
```

```
}
```

Listing A.4: NodeMEI.java

```
package net.ponder2.managedobject;

/**
 * imports removed to save space
 */
public class NodeMEI extends LocalMEI{
private static Logger logger =
        Logger.getLogger(NodeMEI.class);

/**
 * MEI Metrics
 */
private float cpuUtilization;
private float memoryUtilization;
private String ip;

public float getCpuUtilization() {
        return cpuUtilization;
}

@Ponder2op("memoryUtil")
public float getMemoryUtilization() {
        return memoryUtilization;
}

@Ponder2op("getIP")
public String getIp() {
        return ip;
}

@Ponder2op("setIP:")
public void setIp(String ip) {
        this.ip = ip;
}

public boolean helpReq = false;

/**
 * Constructor
```

```
 */
public NodeMEI(String name, String amName) {
        super(name, amName);
}

/**
 * Private and protected methods (Helper methods)
 */
@Override
protected void refreshMetricValues() {
try {
RMIReceiveInterface pt =
        (RMIReceiveInterface) Naming.
                lookup("rmi://"+ip+"/"+this.getAmName());
memoryUtilization =
        pt.executePonderTalk("root/am/system␣memUtil").asFloat();
cpuUtilization =
        pt.executePonderTalk("root/am/system␣cpuUtil").asFloat();
} catch (Exception e) {
logger.error("Getting␣MemUtil␣from:␣rmi://"+ip+"/"+
                this.getAmName());
                e.printStackTrace();
                memoryUtilization = 0;
                cpuUtilization = 0;
        }
}

}
```

Listing A.5: SystemMEI.java

```
package net.ponder2.managedobject;

/**
 * imports removed to save space
 */

public class SystemMEI{

private static Logger logger = Logger.getLogger(SystemMEI.class);

/**
 * MEI Metrics: All other MEOs
```

```
 * in this system
 */
private HashMap<String, LocalMEI> managedObjects;


@Ponder2op("create")
public SystemMEI() {
managedObjects = new HashMap<String, LocalManagedObject>();
PropertyConfigurator.configure("src/resource/logger.properties");
}


@Ponder2op("remove:")
public void removeManagedObject(String name) {
        managedObjects.remove(name);
}

@Ponder2op("refresh")
public void refreshAll(){
for (LocalManagedObject mo : managedObjects.values()) {
                mo.refreshMetricValues();
}
}

/**
 * MEI Actions
 */

/**
 * Greedy approach: find the least busy node.
 * @return NodeName
 */
@Ponder2op("findBestNode")
public String finsBestNode(){
float minMemUtil = Float.MAX_VALUE;
String bestNode = "";
for (LocalMEI mo : managedObjects.values()) {
if(mo instanceof NodeMEI){
        if(((NodeMEI) mo).getMemoryUtilization()
                        < minMemUtil ){
                minMemUtil =
                        ((NodeMEI) mo).getMemoryUtilization();
```

```
                bestNode =
                        ((NodeMEI) mo).getName();
        }
}
}
logger.info("The␣best␣node␣to␣mirate␣to␣is:" + bestNode);
return bestNode;
}

@Ponder2op("sendMigrationNotifyMsg:bestNode:")
public void sendMigrationNotifyMsg(String toNode, String bestNode){
String ponderTalkString = "root/event/MigrationEvent␣create";
try {
        String p2xml = P2Compiler.parse(ponderTalkString);
        P2Object value = new XMLParser().execute(
                        SelfManagedCell.RootDomain, p2xml);
} catch (Exception e) {
        e.printStackTrace();
}
}

/**
 * Greedy Approach: find the least busy VM to migrate.
 * @return
 */
@Ponder2op("findBestVM")
public String findVMToMigrate(){
float minMemUtil = Float.MAX_VALUE;
String bestVM = "";
for (LocalMEI mo: managedObjects.values()) {
if(mo instanceof VirtualMachineMEI){
if(((VirtualMachineMEI) mo).getMemoryUtilization()
                < minMemUtil ){
        minMemUtil =
                ((VirtualMachineMEI) mo).getMemoryUtilization();
        bestVM = ((VirtualMachineMEI) mo).getName();
}
}
}
logger.info("The␣best␣VM␣to␣mirate␣is:" + bestVM);
return bestVM;
}
```

```
@Ponder2op("migrate:To:")
public void migrate(String vmName, String destIP){
if(vmName == null || vmName.isEmpty()){
        logger.error("No␣VM␣has␣chosen␣to␣be␣migrated.");
        return;
}
String destURI = "qemu+ssh://" + destIP
        +"/system␣--migrateuri␣tcp://" + destIP + ":49154";
if(Common.executeCommand("virsh␣-c␣qemu:///system␣migrate␣--live"
        + vmName + "␣" + destURI) == null){
        logger.error("Could␣not␣migrate␣" + vmName
                        + "␣To␣"+destURI);
        return;
}else{
        logger.info("Successfully␣migrated␣" + vmName
                        + "␣To␣"+destURI);
}
}
}
```

Appendix B

Technology Scripts

Listing B.1: KVM.rb

```
#!/usr/bin/ruby

def hasKVM
        if `which virsh`.include?("no␣virsh␣in")
                return "no"
        else
                return "yes"
        end
end

def listVMs
        return [] if not hasKVM()
        return `virsh list`.split("\n")[2..-1].
        map{|line| line.strip().split("␣")[1]}

end


command = ARGV[0]
puts hasKVM() if command == "isvalid"
puts listVMs() if command == "childrenlist"
```

Listing B.2: Ubuntu.rb

```
#!/usr/bin/ruby

def isUbuntu
        if `lsb_release -a 2>/dev/null | grep ID`.
```

```
                split(":")[1].strip()=="Ubuntu"
                return "yes"
    else
                return "no"
    end
end

def listChildren
      return []
end


command = ARGV[0]
puts isUbuntu() if command == "isvalid"
puts listChildren() if command == "childrenlist"
```

Curriculum Vitae

Name:	Omid Mola
Post-Secondary Education and Degrees:	Amirkabir University of Technology Tehran, Iran 2001 - 2005 B.Sc. Amirkabir University of Technology Tehran, Iran 2005 - 2007 M.Sc. University of Western Ontario London, ON, Canada 2008 - 2013 Ph.D.
Related Work Experience:	Teaching Assistant The University of Western Ontario Sep. 2008 - Sep. 2011 Lecturer CS1026, CS1027, CS3342 The University of Western Ontario Sep. 2011 - Dec. 2012

Publications:

- O. Mola, M. Bauer, "Policy-Based Autonomic Collaboration for Cloud Management", ICCGI , pp. 288-293, June 2012 (Best paper award).
- M. Bauer, N. S. McIntyre, N. Sherry, J. Qin, M. Suominen

Fuller, Y. Xie, O. Mola, D. Maxwell, D. Liu, and E. Matias. "Experimenter's portal: the collection, management and analysis of scientific data from remote sites", In Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science (MGC '12). ACM, NY, USA, 2012.

- N. Sherry, J. Qin, M. S. Fuller, Y. Xie, O. Mola, M. Bauer, N. S. McIntyre, D. Maxwell, D. Liu, E. Matias, and C. Armstrong, "Remote internet access to advanced analytical facilities: A new approach with web-based services," Analytical Chemistry Journal, 2012.

- O. Mola, M. Bauer, "Towards Cloud Management by Autonomic Manager Collaboration", International Journal of Communications, Network and System Sciences, special issue on cloud, Dec 2011.

- O. Mola, M. Bauer, "Collaborative Policy-Based Autonomic Management in a Hierarchical Model", IEEE CNSM, October 2011.

- O. Mola, P. Emamian, M. Razzazi, "A Vector Based Algorithm for Semantic Web Services Ranking", IEEE, Proc. of ICTTA08, 2008.

- O. Mola, M. Razzazi, "Design and Architecture of a Search Engine for Grid Services Discovery", 2nd International Conference on Information Systems Technology and Management (ICISTM), 2008.

- M. Razzazi, T. Ghasemi, O. Mola, H. Ghasemalizadeh, "A New Solution For Generalized Search in Grid Environment", IEEE,

Proc. of ICTTA06, P. 1233, April 2006.