ACHIEVING SELF-MANAGED DEPLOYMENT IN A DISTRIBUTED ENVIRONMENT VIA UTILITY FUNCTIONS

by

Debzani Deb

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April, 2008

APPROVAL

of a dissertation submitted by

Debzani Deb

This dissertation has been read by each member of the dissertation committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Professor John Paxton

Approved for the Department Computer Science

Professor John Paxton

Approved for the Division of Graduate Education

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

In presenting this dissertation in partial fulfillment of the requirements for a doctoral degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. I further agree that copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for extensive copying or reproduction of this dissertation should be referred to ProQuest Information and Learning, 300 North Zeeb Road, Ann Arbor, Michigan 48106, to whom I have granted "the exclusive right to reproduce and distribute my dissertation in and from microform along with the non-exclusive right to reproduce and distribute my abstract in any format in whole or in part."

Debzani Deb

April, 2008

To my parents

ACKNOWLEDGEMENTS

I would like to thank my previous advisor, Prof. Michael J. Oudshoorn, whose expert guidance helped me navigate through my research over the years, and whose constant supervision and encouragement enabled me to accomplish my work.

I would like to express my gratitude to my current advisor, Prof. John Paxton for his unparalleled support and encouragement that became so important at the end of my program. His constant emphasis on research quality, insightful comments and feedback helped me maintain a high standard in my work.

My profound gratitude to my doctoral committee members for the great interest they showed in my work, their constructive suggestions and recommendations, and their continuous encouragements. I am thankful to the Benjamin foundation for awarding me a Ph.D. Fellowship. I would also like to thank Montana NASA EPSCoR for providing financial assistance that supported my dissertation work. I am also very thankful to the department's staff members, particularly Jeannette Radcliffe, Kathy Hollenback and Scott Dowdle for their valuable assistance.

I am very appreciative of the love and support that I received from all my friends in MSU and Bozeman. My heartfelt gratitude goes to my parents and sisters for their blessings, love and inspiration. Most importantly, getting to this stage would have been impossible without the support and companionship of my husband Fuad. My eight month old daughter Adrita has been a great inspiration for me. I thank them for the sacrifices they made for me and for being a great source of comfort and affection.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF FIGURES

Figure Page

LIST OF FIGURES - CONTINUED

Figure Page

ABSTRACT

This dissertation presents algorithms and mechanisms that enable self-managed, scalable and efficient deployment of large-scale scientific and engineering applications in a highly dynamic and unpredictable distributed environment. Typically these applications are composed of a large number of distributed components and it is important to meet the computational power and network bandwidth requirements of those components and their interactions. However satisfying these requirements in a large-scale, shared, heterogeneous, and highly dynamic distributed environment is a significant challenge. As systems and applications grow in scale and complexity, attaining the desired level of performance in this uncertain environment using current approaches based on global knowledge, centralized scheduling and manual reallocation becomes infeasible.

This dissertation focuses on the modeling of the application and underlying architecture into a common abstraction and on the incorporations of autonomic features into those abstractions to achieve self-managed deployment. In particular, we developed techniques for automatically identifying application components and their estimated resource requirements within an application and used them in order to model the application into a graph abstraction. We also developed techniques that allow the distributed resources to self-organize in a utility-aware way while assuming minimal knowledge about the system. Finally, to achieve self-managed deployment of application components to the distributed nodes, we designed a scalable and adaptive scheduling algorithm which is governed by a utility function. The utility function, which combines several application and system level attributes, governs both the initial deployment of the application components and their reconfigurations despite the dynamism and uncertainty associated with the computing environment.

The experimental results show that it is possible to achieve and maintain efficient deployment by applying the utility function derived in this paper based solely on locally available information and without costly global communication or synchronization. The self-management is therefore decentralized and provides better adaptability, scalability and robustness.

CHAPTER 1

INTRODUCTION

The growth of the Internet, along with the proliferation of powerful workstations and high speed networks as low-cost commodity components, is revolutionizing the way scientists and engineers approach their computational problems. With these new technologies, it is possible to aggregate large numbers of geographically distributed computing and communication resources with diverse capacities into a large-scale integrated system. Many scientific fields, such as genomics, astrophysics, geophysics, computational neuroscience and bioinformatics which require massive computational power and resources, can benefit from such an integrated infrastructure.

In most corporations, research institutes or universities, significant amounts of computing resources are underutilized at various times. By harnessing the computing power and storage of these idle or underutilized resources, a large-scale computing environment with substantial power and capacity can be created. With such an infrastructure, it is possible to solve computationally intensive problems efficiently and cost effectively than with more expensive, traditional computational resources such as supercomputers. This is also the motivating factor for many of the today's emerging concepts such as Peer-to-Peer (P2P) Computing [1], Software Agents [2], Internet Computing [3], and Grid Computing [4]. However, managing such an integrated infrastructure is a significant challenge even to the most skilled professionals due to the dynamism, unpredictability and heterogeneity associated with it.

The prospect of such a large-scale integrated system has led to a proliferation of the large-scale, high performance scientific and engineering applications. In general these applications are composed of a large number of distributed components and it is important to deploy them in the underlying network in a way that meets the computational power and network bandwidth requirements of those components and their interactions. To accomplish this, it is important to estimate the computational and communication resource requirements of the application components and their links so that sufficient and appropriate resources can be allocated to each application. Moreover, this deployment must ensure efficient utilization of system resources in order to maximize the overall utility of the system.

Satisfying these requirements in a large-scale, non-dedicated, heterogeneous, and highly dynamic distributed environment is a significant challenge as the operating entities may vary dynamically in a short period of time, and in order to maintain the desired performance the computing environment may need to be re-organized. Furthermore, the complexity and cost associated with the management of such an infrastructure and the necessary enhancement of an existing application so that it deploys effectively on that dynamic infrastructure are significant. As the operating environment and applications grow in scale and complexity, attaining the desired level of performance in this dynamic environment becomes infeasible using current approaches that are based on global knowledge, heuristics, centralized scheduling and manual allocation.

This dissertation addresses these challenges by proposing algorithms and mechanisms for achieving *self-managed deployment* of computationally intensive

scientific and engineering applications within a highly dynamic and unpredictable distributed environment. We define self-managed deployment to be one that:

- Automatically identifies the application components and their interactions, along with their estimated resource requirements within an application.
- Automates the task of deploying the application components and their interactions across a large-scale integrated infrastructure in a way that meets their performance requirements and maximizes overall resource utilization of the system.
- Dynamically adapts to changes in the operating environment.

Such automation of the deployment process is paramount in order to reduce operation costs, to allow developers to largely ignore complex distribution and coordination issues, to manage system complexities, to maximize overall utilization of the system and to increase system robustness by dynamically adapting to a changing execution environment.

The goal of this dissertation is to achieve a self-managed, cost-effective, scalable deployment by offering spare compute cycles to scientific and engineering research groups to satisfy their various demands (performance, adaptability, scalability, robustness and so on) and, at the same time, maximizing the use of existing resources. We envision the problem as an autonomic computing challenge [5,6] and our solution incorporates autonomic entities [7] to handle the complexities associated with distribution, configuration, coordination and efficient execution of application components and to adapt to the changes in the underlying environment.

The increasing complexity of computing systems and their interactions with the physical world gives rise to the need for systems capable of self-management. Autonomic Computing aims at realizing computing systems and applications capable of managing themselves with minimum human intervention [8]. An autonomic system, modeled after the autonomic nervous system, refers to a system that is able to protect itself, recover from faults, reconfigure in reaction to changes in the environment and always maintain its operations at a near optimal performance, all with minimal human intervention. In other words, a self-managing system should be self-configuring, self-optimizing, self-healing and self-protecting [9].

The self-managed deployment architecture presented in this dissertation is part of the project ADE, Autonomic Distributed Environment [10, 11, 12, 13], a self-managing distributed system that aims to achieve all four self-* properties while distributing an existing centralized application in a large-scale computing environment. However, this dissertation focuses only on two aspects of self-management: self-configuration and self-optimization. More specifically, this dissertation is concerned with identifying application components and deploying and executing those components across a self-organized network to maximize the overall system utility. Issues such as reliability and security are beyond the scope of this dissertation.

The rest of this chapter is organized as follows. The next section discusses the challenges that needed to be met to achieve self-managed deployment. The subsequent section summarizes the research contributions of this dissertation. The section after that

discusses the target application domains and the final section presents an outline of this dissertation.

Challenges in Achieving Self-Managed Deployment

High performance applications are playing an increasingly important role in the scientific and engineering research community. Typically these applications are inherently multi-phased and highly dynamic and are composed of a large number of distributed components along with dynamic interactions between the components. The distributed infrastructure considered for this research is similarly heterogeneous and dynamic, formed by harnessing the spare compute cycles of distributed computation and communication resources. The combination of these two results in tremendous complexities in application development, distribution, configuration and management. We now explore some of these challenges.

- *Distributed Application Development:* For an average user, the task of effectively partitioning large-scale applications into several components as well as the mapping and scheduling of those components over the distributed and heterogeneous resources is significantly tedious and cumbersome. To transform a regular, centralized application into a distributed one, the programmer needs to perform a large number of changes, most of which require thorough knowledge of both the application structure and the underlying architecture where the application will be deployed.

Distributing an application involves splitting up the functionality of an application into independent or communicating entities so that they can execute in a distributed fashion while minimizing the interactions among the entities. For example, consider the scenario of a large simulation-based application written for a centralized environment that collects some data from several sensors, runs some simulation on that data, analyzes and transforms the simulated results (if necessary) and then visualizes the results. Currently, if the programmer wishes to execute this application on a heterogeneous distributed system he needs to undertake a sequence of complicated activities to ensure efficient distributed execution of the application. First, the programmer needs to manually identify the partitions (e.g. sensor, simulation, analysis, visualization) in the code and then understand how the data is exchanged among the partitions. Then, the programmer needs to decide where to place the partitions relative to the available resources so that the overall communication and latency is minimized. After that, the application needs to be rewritten to incorporate middleware mechanisms to handle the communication and coordination between heterogeneous nodes. Each of the above steps requires a level of skill that most programmers lack. To overcome this difficulty, a self-managed approach should be able to analyze an existing application, determine application components and their interactions automatically along with their estimated resource requirements, distribute them in the underlying infrastructure and coordinate their executions transparently at runtime.

- *Resource Allocation*: Application performance can vary significantly depending on various resource assignments. When an application is deployed across a distributed collection of heterogeneous machines, an effective resource allocation mechanism allocates a set of resources that meets the application's specific needs. For instance, in the case of the above simulation example, perhaps the most effective way to distribute the application is to let the sensors execute on moderately loaded and less powerful standalone nodes while allowing the simulation to execute on a high performance server and the multiple visualization components run on lightly loaded, powerful machines with high-end graphics. Additionally, depending on the interdependencies among these components, resources should be allocated in such a way that the overall communication cost is minimized.

  Traditional manual resource allocation approaches require the application developers to allocate resources to various applications based on the estimation of their resource requirements. However, estimating the resource requirements of an application and mapping the components to the resources accordingly is difficult and time consuming for application developers. This difficulty is usually addressed by over-provisioning of resources, where each application is allocated enough resources to handle its worst-case requirements. Such resource over-provisioning leads to significant under-utilization of the resources. Nowadays, with reduced IT budgets, the expectation is to achieve more with even fewer resources, and, over-provisioning is no longer an option. Moreover, as the current workload of the

available resources may vary and the network configuration may change over time, resources need to be dynamically re-allocated and re-distributed among all executing applications to achieve high resource utilization. To deal with these issues, an automated approach should choose the right set of resources for each application and its components to satisfy the application's configuration and performance requirements and to avoid scarcity of resources.

- *Scalability*: While considering large-scale infrastructure and applications, it is important to avoid any dependence on centralized entities and global knowledge. Decentralized approaches should be incorporated to avoid communication bottlenecks, to keep applications scalable, and to dynamically utilize resources.
- *Adaptability*: The proposed infrastructure is highly dynamic and continuously evolving during the lifetime of an application. This includes the availability and the load on computing resources, changes in network conditions, etc. Such a computing environment thus becomes highly uncertain in terms of the performance and availability of the resources. To deal with these dynamic and unpredictable behaviors, it is crucial to support dynamic adaptations, such as repartitioning, reallocation and reconfiguration.

Summary of Research Contributions

In this dissertation, we have addressed several important problems that arise in the design of self-managed deployment. In particular, we developed techniques for automatically identifying application components and their internal dependencies, along

with their estimated resource requirements within a centralized application, and used them to model the application via a graph abstraction. We also developed techniques that allow the distributed resources to organize in a utility-aware tree structure while assuming minimal knowledge about the system. Finally, to achieve self-managed deployment of application components across the network nodes, we have designed a scalable and adaptive scheduling algorithm that is governed by a utility function [14]. The utility function, which combines several application and system level attributes, governs both the initial deployment of the application components and their reconfigurations. This multi-attribute function is carefully formulated to return a scalar value, which signifies the system's overall utility for each possible state of a system. The goal then becomes selecting a state that maximizes the overall utility. During execution, resource allocation and other operating conditions may change; the corresponding change in the overall utility of the system is calculated by this utility function and reconfiguration decisions are made toward maximizing this value. As a result, execution efficiency is maintained despite the dynamism and uncertainty associated with the application and the networked environment.

The self-managed deployment is novel in its consideration of the overall system utility (combining multiple application and system level attributes) during deployment and reconfiguration. The approach is transparent to the end user as it hides the distribution and communication aspects from the programmer by automatically identifying the program components and their interactions within an application and by generating code for their distributed execution. The approach is scalable in its use of a

decentralized and minimal knowledge-based scheduling algorithm to find the highest utility association among the set of application components and the set of distributed nodes. Finally the approach is adaptive as it utilizes decentralized monitoring of distributed resources and incorporates the monitored data into the utility function to trigger reconfiguration. Our experimental results demonstrate that applications with various performance requirements, in terms of network topology, processing and communication needs, are efficiently deployed and managed and at the same time resource utilization is maximized.

Target Application Classes

It is impractical to expect that all centralized applications can be distributed without rewriting their source codes. We target high performance scientific and engineering applications, where the computation-to-communication ratio is significant and partitioning tasks across the network increases performance. These applications typically make significant computation demands that are difficult to meet with a set of co-located computer clusters. Adaptive and distributed execution environments make these computations easier and also create the potential to improve performance via dynamic adaptation. More specifically, this dissertation targets two classes of applications as follows:

- *Master-Worker applications*: These applications are characterized by a large number of computationally intensive tasks independent from each other, i.e. where no inter-task communications are needed and where the tasks can be computed in

any order. The easily decomposable nature and the absence of intranode communication make them feasible for large-scale infrastructure, where such applications of unprecedented scale can be supported. Examples of this type of application include SETI@home [15], biological sequence comparisons [16], factoring large numbers [17], genomics [18], computational neuroscience [19], etc. Many practical approaches exist that efficiently deploy such independent task applications in a distributed setting. However, few of them address the added complexity, due to the dynamic and uncertain characteristics of the operating environment. Typically, the main requirement of this type of application is to maximize the number of tasks computed per time unit, i.e. minimizing the overall execution time.

- *Inherently distributed applications*: These are applications that must execute in a distributed setting to fulfill their functional constraints. Often these applications execute sequentially; however, the goal is to provide a distribution that meets the constraints with minimum communication overhead. For example, consider the simulation application described earlier in this chapter, in which the simulation code executes on a high-performance computing resource and must interact at runtime with services like monitoring, analysis, visualization, archiving and collaboration. A large number of scientific and engineering applications can be characterized by this notion. In addition to the requirement of minimizing the communication overhead of the executing simulation, another key requirement of

these applications is the support for reliable, high-throughput, low latency information flows between the different distributed components of the application.

Outline of the Dissertation

This dissertation is organized as follows. Chapter 2 provides background information on important concepts related to this research and also discusses some of the representative related work found in the literature. Chapter 3 presents the three-layer software architecture designed to achieve self-managed deployment. It also presents a detailed discussion of the suggested mechanism that should be built into systems to enable self-management. Chapter 4 is concerned with the static analysis of the application to extract a graph model for an application. Chapter 5 details the utility-aware self-organization of the network nodes. Chapter 6 illustrates the formulation of the utility function followed by the detailed discussion of initial deployment and optimization. Chapter 7 presents the experimental evaluation of the self-managed deployment architecture. Chapter 8 provides concluding remarks and directions for future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

This dissertation addresses several important issues such as self-management, automatic program partitioning, resource allocation, decentralization and dynamic behavior in an integral way. As a result, it draws on many approaches and techniques practiced in research fields such as Autonomic Computing, Automatic Program Partitioning and Distribution, Resource Management, Adaptive Systems, etc. and becomes relevant to the research projects in these areas. The following sections detail the basics of these research fields along with the relevant research projects.

Autonomic Computing

Autonomic Computing is a relatively new idea that focuses on delivering self-managing computer systems; systems that regulate themselves much in the same way our autonomic nervous system regulates and protects our bodies [7]. With choosing the term *autonomic*, researchers attempted to provide self-managing capabilities in computer systems with the aim of decreasing the cost of developing and managing them. An Autonomic Computing System is essentially a collection of Autonomic Elements (AE), which can manage their internal behaviors as well as the relationship with others in accordance with high-level policies from a human administrator [5]. At its core, an Autonomic System should posses the following four essential properties:

- *Self-configuring capabilities*: An autonomic system must be able to configure and reconfigure itself under varying and unpredictable conditions. An autonomic system must have the ability to dynamically adjust its resources based on its state and the state of its execution environment.
- *Self-optimizing capabilities*: An autonomic system must provide operational efficiency by tuning resources and balancing workloads. Such a system must continually monitor itself to detect performance degradation in system behaviors and intelligently perform self-optimization functions to meet the ever-changing needs of the application environment. Capabilities such as repartitioning, load balancing, and rerouting must be designed into the system to provide self-optimization.
- *Self-healing capabilities*: An autonomic system must provide resiliency by discovering and preventing disruptions as well as recovering from malfunctions. Such a system must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction. It must be able to discover problems or potential problems that might cause service disruptions, and then find an alternative way of using resources or reconfiguring the system with minimal loss of information or delay, and no loss of integrity.
- *Self-protecting capabilities*: Self-protecting systems secure information and resources by anticipating, detecting, and protecting against various types of attacks such as viruses, unauthorized access, and denial-of-service. Such a system

must be able to maintain the overall security and integrity through the use of pattern recognition and other techniques.

Automating the management of the computing resources is not a new idea to be explored by computer scientists. For decades, software systems have been evolving to deal with the increased complexities associated with system control, adaptation, resource sharing, and operational management. Research in many isolated fields such as security, fault tolerance, artificial intelligence, human computer interaction, networking, software agents, dynamic resource management, etc. has delivered systems that managed to handle the complexities in their own domains. Autonomic computing is just the next logical evolution of these past trends to address today's increasingly complex and distributed computing environments in an integral way and by using the notion of self-management, the idea is to unite these disciplines into a single field named Autonomic Computing.

IBM was the first to coin the term "autonomic" to describe systems capable of managing themselves [7]. Several other major IT companies have developed their own initiatives for promoting self-adaptable systems. For example, the Adaptive Enterprise initiative by HP [20] and the Dynamic Systems initiative by Microsoft [21]. There are initiatives at a smaller scale including the Autonomic Platform Research [22] by Intel and Automatic Workload Repository (AWR) for database self-management [23] by Oracle.

Architectural Framework for Autonomic Systems

IBM researchers have established an architectural framework for autonomic systems [7]. An autonomic system consists of a set of Autonomic Elements that contain and manage resources and deliver services to humans or other autonomic elements. An

autonomic element consists of one autonomic manager and one or more managed elements. At the core of an autonomic element is a control loop that integrates the manager with the managed element. The autonomic manager consists of sensors, effectors, and a five-component analysis and planning engine as depicted in Figure 1 [12]. The monitor observes the sensors, filters the data collected from them, and then stores the distilled data in the knowledge base. The analysis engine compares the collected data against the desired sensor values also stored in the knowledge base. The planning engine devises strategies to correct the trends identified by the planning engine. The execution engine finally adjusts parameters of the managed element by means of effectors and stores the affected values in the knowledge base.

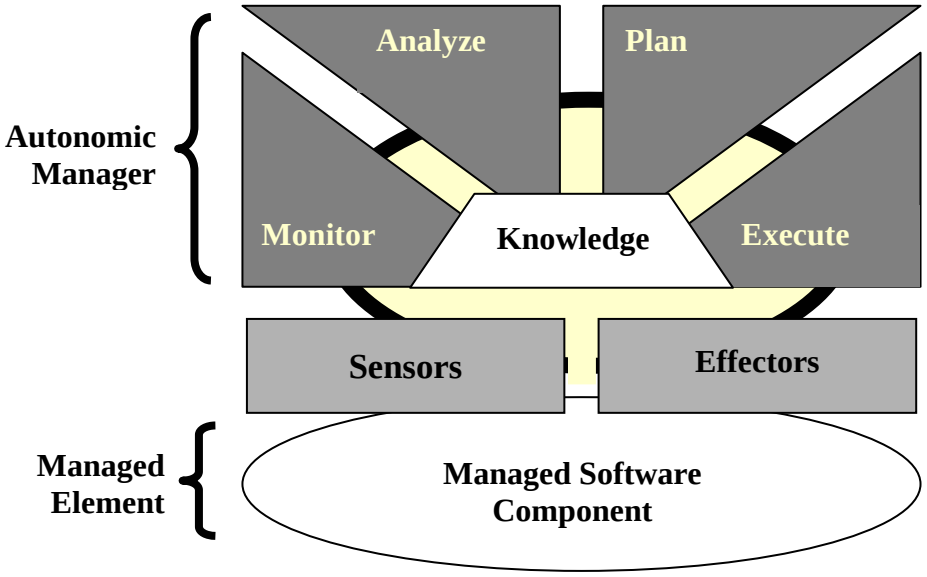


Figure 1. The Structure of an Autonomic Element.

An autonomic element manages its own internal state and its interactions with its environment (i.e., other autonomic elements). An element's internal behavior and its relationships with other elements are driven by the goals and policies the designers have

built into the system. One of the key values of autonomic computing is that the interface between the autonomic manager and the managed element are standardized. That is, a single standard manageability interface can be used to manage routers, servers, application software, middleware, a web service, or any other autonomic element. This single manageability interface constitutes a level of indirection and is the key to adaptability.

Policies and Utility Function

Policy plays an important role in autonomic computing as they actually guide the behavior of the system. Autonomic elements can function at different levels of behavioral specification. At the lowest levels, the capabilities and the interaction range of an autonomic element are limited and hard-coded. At higher levels, autonomic elements pursue more flexible goals specified with policies, and the relationships among elements are flexible and may evolve.

Recently, Kephart and Walsh proposed a unified framework for policies based on the well-understood notions of states and actions [24]. In this framework, a policy directly or indirectly causes an action to be taken that transforms the system into a new state. Kephart and Walsh distinguish three types of policies useful for Autonomic Computing, from the lowest to the highest level of behavioral specification. At the lowest level, an action policy determines the action that should be taken when the system is in either a specific state or a set of possible states that all satisfy a given condition. In the next level, a goal policy specifies either a single desired state, or one or more criteria that characterize an entire set of desired states and let the system itself determine how to

achieve the desired state. Rather than specifying actions to be taken at each and every state, the goal policy generates rational behavior by using sophisticated planning or modeling algorithms. However, even in the goal policy, the system administrator/policy writer has to identify and specify the desired states explicitly.

Utility function policies define an objective function that aims to model the behavior of the system at each possible state. Rather than explicitly specifying desirable states, they ascribe a real-valued scalar desirability to each state. The most desired state is then computed on a recurrent basis by selecting the state that has the highest utility from the present collection of feasible states. Utility function policies provide more fine-grained and flexible specification of behavior than goal and action policies. In situations in which multiple goal policies would conflict (i.e., they could not be simultaneously achieved), utility function policies allow for unambiguous, rational decision making by specifying the appropriate tradeoff.

Relevant Research Projects

Since Paul Horn introduced Autonomic Computing Systems to the National Academy of Engineering at Harvard University in 2001, IBM has been extremely successful in rallying the research community behind their autonomic computing initiative. Fully automating the organization and optimization of a large distributed system is a staggering challenge and there are numerous research groups working towards this goal. While there are numerous studies that have addressed all four or some aspects of self-management, this section only discusses the autonomic research

approaches related to this research, i.e. those that focus primarily on self configuration and self optimization.

Accord [24, 25] is a component-based programming framework to support the development of autonomic application in grid environments [4]. Their research is motivated by the observation that the underlying grid computing environment is inherently large, complex, heterogeneous and dynamic and so are the emerging applications that exploit such a system. In their work, they have proposed a new programming paradigm where the composition (configuration, interaction and coordination) aspect is separated from computations in component/service-based models and both computations and compositions can be dynamically managed by the rules that are introduced during runtime.

The overall objective of the AutoMate [26] project is to investigate key technologies to enable the development of autonomic grid applications that are context aware and capable of self-configuring, self-composing, self-optimizing, and self-adapting. More specifically, it investigates the definition of autonomic components, the development of autonomic applications as dynamic compositions of autonomic components, and the design of key enhancements to existing grid middleware and runtime services to support these applications. AutoMate develops an autonomic composition engine to calculate a composition plan of components based on dynamically defined objective constraints that describe how a given high-level task can be achieved by using available basic grid services.

The goal of the Autonomia [27] project is to develop an infrastructure and tools that provide dynamically programmable control and management services to support the development and deployment of autonomic applications. Their prototype implements two important properties of autonomic systems: self-configuration and self-healing by using a mobile agent system.

The aforementioned approaches [25, 26, 27] are developing environments and suggest new programming metaphors in order to realize the desired benefits of self-management in a distributed environment. In other words, these approaches are well suited for the development of new autonomic applications, but they cannot be applied to inject self-managing behavior into existing applications. On the contrary, this dissertation proposes an execution environment intended for transforming applications written in a centralized fashion to the corresponding distributed version and executing it in a networked environment while offering self-management at both the application and network level.

Unity [28] is a research project, undertaken at IBM's Thomas J. Watson Research Center, that investigates some of the methodologies and technologies that allow a complex distributed systems to be self-managed. In their prototype environment, the self-management is achieved via interconnections amongst a number of autonomous agents. The major autonomic computing aspects realized by Unity are 1) the way the system uses goal-driven self-assembly to configure itself at runtime, 2) the use of the utility function to maximize the system's overall business objectives and 3) the design patterns that enable self-healing within the system. However, the approach does not scale well with

respect to the number of applications and with respect to the number of resources in the system as they compute a globally optimal allocation of servers across the application. In contrast, by utilizing a hierarchically organized network model, this research intends to make local optimization decisions that are lightweight and decentralized, and as a result, provides better adaptability and scalability in a dynamic environment.

The goal of the Kinesthetics eXtreme or KX Project [29] is to inject autonomic computing technology into legacy software systems without any need to understand or modify the code of the existing system. More specifically, KX retrofits autonomic capabilities onto legacy systems designed and developed without monitoring and dynamic adaptation in mind. The project designed a four-tiered architecture composed of probe, gauge, controller and effectors to add autonomic services explicitly via an attached feedback loop that provides continual monitoring and, as needed, reconfiguration or redeployment. The lightweight design and separation of concerns enables easy adoption of individual components, as well as the full infrastructure, for use with a large variety of systems. KX is being used to add self-configuration and self-healing functionality to several legacy systems, whereas this dissertation is focused to provide self-organization and self-optimization. In addition to retrofitting autonomic computing, without any need to understand or modify the target system's code as in KX, this research also aims to relieve the programmer from the distribution concerns by automatically transforming a centralized application into a distributed one and deploying it to the underlying network

The Rainbow Project [30] investigates the use of software architectural models at runtime as the basis for reflection and dynamic adaptation. This architectural model

provides a global perspective on the system by revealing all the components (the system's principal computational elements and data stores) and how they connect. The model also contains important properties such as each server's load, each connection's bandwidth, and the response time experienced by each client. This architectural model is used to monitor and reason about the system and therefore self adaptation holds. The project aims to provide capabilities that will reduce the need for user intervention in adapting systems to achieve quality goals, improve the dependability of changes, and support a new breed of systems that can perform reliable self-modification in response to dynamic changes in the environment. Rainbow only abstracts the underlying infrastructure to provide self-adaptation, whereas this dissertation aims to model both the application and the underlying network into a common abstraction in order to achieve self-managed deployment. Furthermore Rainbow is inherently centralized, with monitoring and adaptation performed within a single Rainbow instance and therefore is not scalable.

Astrolabe [31] is designed to automate self-configuration and self-monitoring and to control adaptation. Astrolabe operates by creating a virtual system-wide hierarchical database of the state of a collection of distributed resources, which evolves as the underlying information changes. A novel peer-to-peer protocol is used to implement the Astrolabe system, which propagates the updates within seconds, even in large networks and operates without any central servers. Astrolabe is secure and robust under a wide range of failure and attack scenarios, and it imposes low loads even under stress.

The AutoFlow [32,33] project aims to develop a self-adaptive middleware high-end enterprise and scientific applications. Applications are abstracted into an Information Flow Graph consisting of source, sink, flow operators and edges. Once a flow graph is described, its deployment creates an overlay across the underlying physical distributed system. Automated methods for initial deployment and runtime reconfiguration are based on resource awareness and on utility functions. AutoFlow scales to large underlying platforms by using hierarchical techniques for autonomic management, which allow decentralized decision making rather than a globally optimal decision that involves costly communication among the nodes. To achieve their goal, they adopt a hierarchical organization of underlying resources clustered according to various system attributes such as end-to-end delay, bandwidth, etc. However, calculating and maintaining the clustering structure is time consuming and needs to be repeated if the resource attributes changes over time. An alternative organization of the resources that avoid this shortcoming is presented in Chapter 5 of this dissertation.

None of the abovementioned systems has the same goals as this research. Our objective is to provide self-managed deployment of centralized applications in distributed settings. More specifically, this dissertation focuses on the automatic decomposition of the centralized application into self-managed components and their distribution across the underlying network in order to maximize certain utilities.

Automatic Application Partitioning and Distribution

Automatic partitioning systems are able to automatically partition an existing program, written for a standalone system, into several communicating components. These partitioned components can then be distributed amongst the underlying network nodes. The idea is to be able to reconfigure and redeploy an existing centralized application without rewriting the application's source code or modifying the existing runtime environment. Automatic partitioning can offer several advantages and can greatly simplify the way many distributed systems are developed, such as 1) there is no need of writing complex and error prone code for interprocess communication, 2) applications written in a centralized fashion can be re-partitioned and re-deployed in different distributed settings without modifying the source code, 3) network traffic may be reduced as related program segments are placed near the resources and 4) as the applications are capable of executing within a standard runtime environment; this also allows an easy and efficient deployment of the application on a wide variety of resources.

The idea has been explored in a number of earlier systems, most of which are Java-based. Such systems basically take centralized code in Java as input along with user specified location information for application data and code (e.g. classes, methods, etc.). The original application code is then rewritten so that code and data are divided into components that can be executed in the desired location. Depending upon the user inputs, some of the classes are made remotely accessible by using standard object-oriented (OO) techniques of inheritance and interfaces. Local data exchange in the centralized application (e.g. function calls, data sharing through references and so on) is replaced

with remote communication (e.g. remote function calls using Java RMI, references to a proxy object – that could be pointing either to a local object or to a remote object, etc). Many other additional transformations need to be done to ensure correct execution of the transformed program in distributed settings. The following sections give a brief overview of some of these systems.

Java Party [34] is an extension of Java that automatically transforms regular Java classes into remotely accessible ones. Users specify which objects are to be made remote by annotating their classes with an access modifier (keyword *remote*). The system is designed to have a translator before JVM to generate Java code for remote classes as well as translate the other code referencing it. JavaParty is centralized in the sense that the Runtime Manager runs as a central component, keeping track of all the classes and a local component runs at each node that hosts objects and helps in migration, etc. Object migration is handled by making an explicit call to the central component.

Like JavaParty, Doorastha [35] also allows the user to annotate a centralized program to turn into a distributed application. However, the Doorastha annotations are quite expressive compared to JavaParty annotations and the user has to go through a significant amount of detailed semantic specification to use the system correctly. Doorastha requires the user to annotate – classes, instance fields, methods, and method arguments and specify the call semantics (by reference, copy, etc) for them. These tags are then evaluated by the system to generate the Java bytecode.

Pangaea [36] is an automatic partitioning tool that statically analyzes object-oriented programs and distributes them automatically on a networked system using both

JavaParty and Doorastha as back ends. Pangaea does static analysis of the application source code to determine the *Object Graph* approximating the runtime program structure. With the help of this graph, a user then creates partitions (consisting of groups of objects) and specifies, through a graphical user interface, which partitions are tied to which hosts. This information is then used to generate the code that can directly be fed into an existing partitioning system like Java Party or Doorastha.

Addistant [37] is a Java bytecode translator for automatic distribution of legacy Java applications. It takes a Java application to be partitioned and uses a user specified placement policy to translate it into a distributed version. More specifically, the user needs to write a policy file for specifying where the instances of each class are allocated and how remote references to those instances are implemented. Therefore, the user must have some knowledge of the source code to specify placement policies. Another limitation of Addistant is that it only provides class-based distribution: all instances of a class must be allocated on the same node.

J-orchestra [38] is GUI-based and performs all transformations at the bytecode level. J-orchestra provides the user with a front-end profiler that reports the statistics on the interdependencies of various classes based on profiled runs of the application. Based on this information the user then specifies the mobility properties and location of the classes. For every Java System or application level class involved in the system, the user can specify whether the class instances will be mobile or anchored (classes that contain platform-specific code in native format are considered unmodifiable and should be anchored to their host). For mobile classes, the user needs to provide a migration policy

and for the anchored classes, the user needs to specify their locations. Using this input, J-orchestra then rewrites the application code to create the final distributed application. J-orchestra is the first system to address the problem with unmodifiable code effectively. This is actually done through static analysis of the code that tries to determine heuristically what references lead to unmodifiable code and a sophisticated rewrite mechanism then transforms those indirect references to direct references (and vice-versa) at runtime. The role of the analysis and profiler are strictly advisory, the user may or may not follow the guidance and can override the analysis results at will. Thus, J-orchestra provides the user with tools that automatically infer many of the essential concerns for partitioning and distribution, but the user has complete control and can override the distribution decisions at any time.

Coign [39] is an automatic partitioning system for software based on Microsoft's proprietary Component Object Model (COM) and thereby its applicability is limited to a small range of applications. Coign uses scenario-based profiling to profile communication among components and based on the profiled data, partitions and distributes components among the distributed resources.

Reference [40] automatically translates monolithic applications written in Java bytecode into multiple communicating parts in a networked system. However, it does not give the user any control over distribution.

Systems based on Distributed Shared Memory (DSM) [41,42,43,44] also share the same goal as automatic partitioning. However they use a specialized runtime environment in order to detect access to remote data and to ensure data consistency,

consequently it loses portability. In contrast, automatically partitioned Java application executes on original, unmodified JVMs and thereby is deployable on a wide range of distributed resources supporting JVM.

The main difference between the abovementioned automatic partitioning systems and this research is that our partitioning and distribution decisions are resource aware and utility-driven. Typically, an automatic partitioning system only contains knowledge about the internal structure of the application, not about the environment where the application is going to be deployed. This research, on the other hand, is knowledgeable about both the application and the underlying network and automatically reallocates resources and reconfigures the deployed application graph to maximize the overall utility of the system. Another important difference is the level of transparency provided by our approach compared to the above systems. Some systems rely on the user's knowledge of the application to some extent and require the user to specify the network locations of hardware and software resources and annotate the code using them directly [35,36,37], while others completely rely on the results of their automatic analysis and do not give the user any control at all [40]. However, in this research, control can be entirely automatic (where all the specifications are inferred automatically by the framework), entirely manual (where the end-user is responsible for all specifications), or anywhere in the space in-between. In our approach, user-level control can be exerted by providing high level policies and constraints.

Adaptive Systems

An adaptive software system is capable of changing its behavior dynamically in response to changes in its execution environment. The interest in dynamically reconfigurable software systems has increased significantly during the past decade due to the advances in supporting technologies like computational reflection, aspect-oriented programming and component-based design [45].

One particularly widely used technique for adaptation is aspect-oriented programming (AOP) [46], where the code implementing a crosscutting concern (e.g. quality of service, performance, fault tolerance, security), called an *aspect*, is developed separately from other parts of the system and *woven* with the application code at compile- or runtime. AspectJ [47] and ApectC# [48] are examples of systems that perform weaving at compile time; Weave.NET [49] and Aspect.NET [50] on the other hand perform weaving after compile time but before load time; and A Dynamic AOP-Engine for .NET [51], AspectIX [52] and CLAW [53] perform dynamic weaving at the bytecode level.

A number of adaptive systems provide a programming framework that enables programmers to design, develop and optimize adaptive distributed applications with explicit constructs for adaptation and reconfiguration. Open Java [54], Adaptive Java [55], and PCL [56] are examples of such systems. This approach is well suited for the development of new adaptive applications. However, injecting adaptation into existing non-adaptive programs requires modifying the application source code and therefore this approach can not be applied transparently to existing programs.

Another way of providing adaptive behavior to an application is to use adaptive middleware, usually extensions of popular object oriented middleware platform such as CORBA, Java RMI, DCOM and .NET, where adaptation is incorporated by intercepting and modifying messages passed through the middleware. Examples include TAO [57], Dynamic TAO [58], Open ORB [59], QuO [60], Squirrel [61], and IRL [62]. Component-oriented middleware is a recent evolution of object-oriented middleware which uses the abstraction of a component instead of an object to encapsulate reconfiguration capabilities. Examples include Enterprise JavaBeans [63] and the CORBA Component Model (CCM) [64].

Other approaches implement adaptive behavior by extending the virtual machine to intercept method invocation, object creation and reading and writing operations to a data field during runtime. The intercepted operation is then made adaptable by loading new code dynamically. Examples include Iguana/J [65], metaXa [66], Guarana [67] and PROSE [68]. While this method can add adaptive code transparently to an application, the application needs that specific virtual machine to execute, thus limiting its applicability.

CHAPTER 3

SYSTEM ARCHITECTURE

This chapter first presents the design and architecture of ADE and then elaborates how the main concern of this dissertation – self-managed application deployment fits into the overall flow of operation. It then presents the three-tier architecture specifically designed to realize the self-managed deployment of an application in a distributed environment.

Project ADE: Enabling Self-managed Distribution

ADE [8,9,10,11] aims to achieve self-management of a distributed system via interconnections among autonomic elements across the system. ADE targets existing Java programs, consisting of independent or communicating objects as application components, and automatically generates a self-managed distributed version of that program. Based on the cross-platform Java technology, ADE is expected to support all major contemporary platforms and handle heterogeneous issues successfully. The availability of the source code can not always be assumed, so ADE performs analysis and transformations at the bytecode level. However, applications for which source code is available can be transformed to bytecode and can exploit the benefits offered by ADE.

Flow of Operation in ADE

Figure 2 shows ADE's overall flow of operation [9,10]. At first, a code analyzer statically inspects the user supplied bytecode to derive an object interaction graph. Based

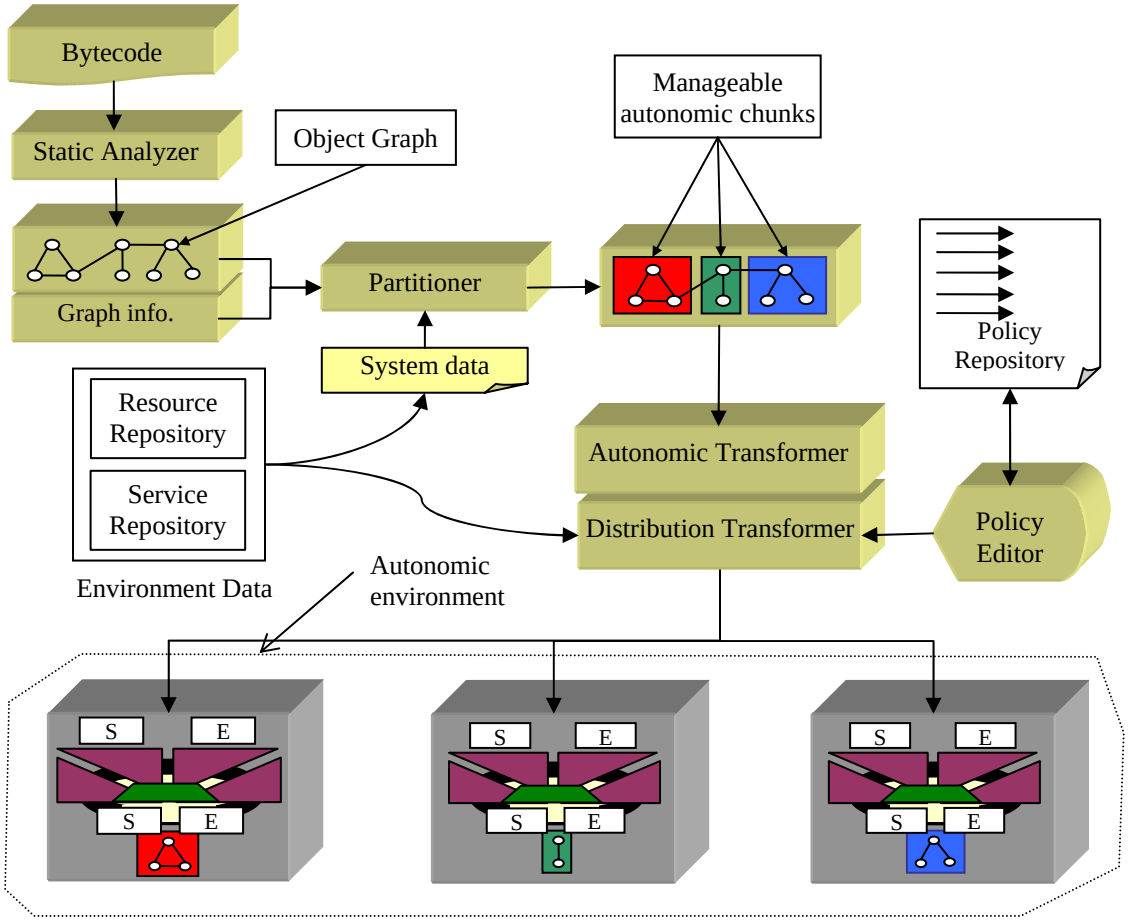


Figure 2. Overall Flow of Operation in ADE.

on this graph, the partitioner then generates several partitions (consisting of a single object or groups of objects) along with the distribution policies and deploys those partitions to a set of available resources. Deployment decisions are based on several criteria such as the resource (CPU, memory, communication bandwidth, etc.) requirement of the objects and their interactions, various system information collected via monitoring services such as resource availability, workload, usage pattern, or any user supplied policy. Deployment is also adaptive such that it has the ability to dynamically reconfigure with respect to the changes in the computing environment. Chapter 6 of this dissertation

illustrates the algorithms and techniques designed to achieve such self-managed deployment.

During deployment, an autonomic transformer injects the distribution and self-management primitives to the partitions according to the system deduced distribution policies and any other user-supplied policies/constraints (e.g. component *C* requires some input data that resides on Machine *M*, so the component *C* should execute on Machine *M*) so that the resultant self-managed partitions can execute on different nodes in a distributed fashion. The underlying system comprises a platform-agnostic language and the associate pre-processor for bytecode to bytecode translation. The transformed program is based on self-contained concurrent objects communicating through any standard communication protocol and incorporates salient features from existing middleware technologies.

Autonomic Elements in ADE

In ADE, every distributed site (i.e. PCs, laptops, workstations, servers, etc.) is managed by an autonomic element that controls resources and interacts with other autonomic elements in the system. More specifically, each autonomic element encapsulates the program partition allocated to the site managed by it as its Managed Element and interacts with the environment by using standard autonomic metaphors. Each autonomic element monitors the actual execution of the application and the behavior of the resource itself. They also set up a mutual service relationship to interact with each other so that information can be shared among them. Based on the information, the underlying autonomic system then adjusts the parameters found by static analysis

such as computational and communication requirements to their runtime values and if needed dynamically repartitions the graph. Besides this basic functionality, some of the autonomic elements in the system are given some higher level management authority such as managing the system registry or policy depository; acting as the user interface for program partitioning and transformation; being the source or destination of program input and output, etc.

To use the autonomic resources, a potential user must first register her computer with the autonomic system through a user portal. Once registered, an autonomic element is initiated on that machine and configures itself properly with all the necessary system data and policy information and consequently makes its services available to other autonomic elements. A user may deregister his/her machine at any time and consequently the autonomic element running on that machine will delegate its current managed element to other available autonomic element without the loss of useful computation. Since the distributed environment is shared by many users, the environment may change at runtime and so does the application's communication pattern, consequently the autonomic element needs to adapt accordingly. To achieve this, the autonomic element provides monitoring services and, based on the monitored data, automatically makes decisions such as migrating the managed element (or portion) to a less busy autonomic element, delaying other non-dedicated tasks to consume more resource, initiating backup to accommodate more tasks, and so on.

Application Deployment in ADE

ADE supports multiple, logically separated application environments each capable of supporting a distinct application. As the application components within an application execute with different constraints and requirements, they should be mapped to appropriate hardware resources in the distributed environment so that their constraints are satisfied and they provide the desired level of performance. Mapping between these resource requirements and the specific resources that are used to host the application is not straightforward.

We designed a three step process to perform this mapping as shown in Figure 3. In the first step of mapping, the application's code is statically analyzed to extract an application model expressed as lower-level resource requirements such as processing, bandwidth, storage, etc. The next step involves constructing a model of the underlying network by obtaining knowledge about available resources such as their computational capabilities, connectivities and workloads and then organizing them according to network proximity (considering latency and bandwidth). The third and final step allocates a specific set of resources to each application with respect to the resources required by the application components and the resources available in the system. The goal of the mapping is to maximize the system's overall utility based on certain policies, priorities, user-defined constraints and environmental conditions. In short, this dissertation focuses on the modeling of the application and underlying architecture into a common abstraction and on the incorporation of autonomic features to those abstractions to achieve self-managed deployment.

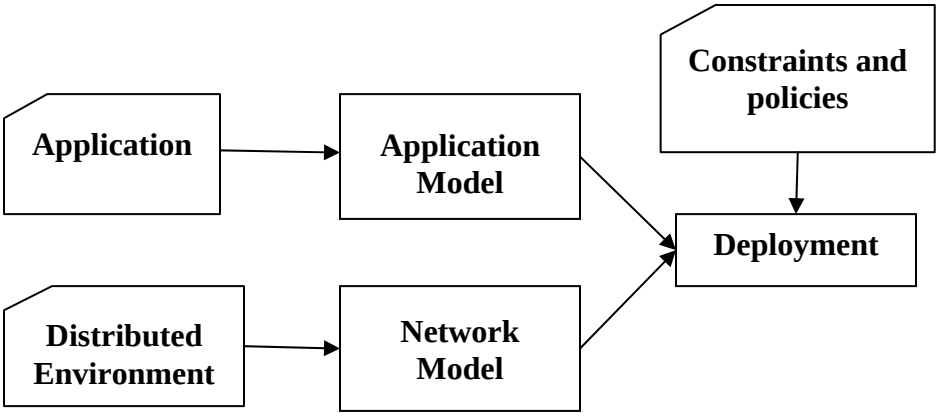


Figure 3. Application Deployment Process in ADE.

Once the application components are automatically deployed into the distributed environment, dynamic reconfiguration is triggered as needed. Based on the observations obtained by the monitoring of the system resources, our approach may automatically migrate application components, reallocate resources and redeploy the application graph. With respect to the self-optimizing criteria, we decided to maximize a specific utility function that returns a measure of the overall system utility based on the executing application's requirements, the system's operating conditions as well as some user policies, priorities and constraints. At any time during execution, the operating conditions change; the corresponding change in the overall utility of the system is calculated by this utility function and the computing environment is reconfigured to a state that maximizes the utility.

Deployment Architecture

Self-managed deployment in ADE is achieved by adopting the three-layer architecture as shown in Figure 4 and described as follows:

- The Application layer is responsible for analyzing the application code and deriving the application model consisting of application components and their resource requirements.
- The Autonomic Layer is responsible for deployment of the application and for ensuring self-managing behavior of the system. This layer is divided among autonomic elements that are responsible for resource monitoring, utility function evaluating, initial deployment considering policies and constraints, reconfiguration, middleware services, etc.
- The Network layer organizes the physical nodes in the network into a utility-aware tree structure and is utilized by the upper level autonomic layer for deployment.

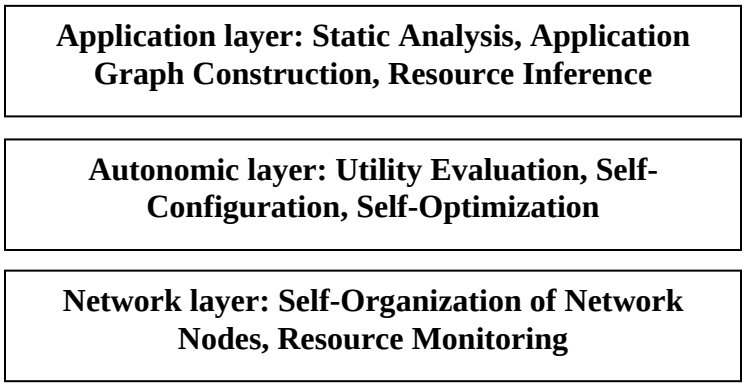


Figure 4. Deployment Architecture.

CHAPTER 4

THE APPLICATION LAYER

This chapter starts by providing a formal definition of the graph abstraction used to model an application in this dissertation. The chapter then presents the detailed description of the static, compile-time analysis that is performed on the application code to extract its estimated runtime structure and resource requirements. The chapter also explains and evaluates each step of our static analysis approach with a representative example. We then discuss some of the limitations of our approach and outline possible improvements. Some of the work presented in this chapter appears in the following publications: [69, 70].

Graph Representation of an Application

To be truly autonomic, a computing system needs to know and understand each of its elements. One such element is the application executing on the autonomic infrastructure; the system needs detailed knowledge of it. Analyzing and representing software in terms of its components and their internal dependencies is important in order to provide self-managing capabilities because this is actually the system's view of the runtime structure of a program. To satisfy application performance goals, it is also important to infer the resource requirement of application components and their links so that an efficient mapping of components to resources can be achieved.

In this dissertation, an application is modeled as a graph consisting of application components and the interactions among them, along with the estimated resource requirements of the components and their links. Having a program's total view in terms of a graph, it is possible to make informative placement and optimization decisions. For instance, at runtime, due to load or some other factors, an Autonomic Element (AE), managing a certain node may decide to migrate its managed element to a less loaded machine; and there may be several machines available to handle that load. As each AE has available to it the complete structure of the application graph and the information about which partitions are managed by which AEs (obtained via monitoring services and interaction among AEs), the optimal replacement AE can be found that is closest to other AEs managing other partitions that have active communication with the partition to be migrated. Therefore, an application graph is not only important for initial configuration of the components to the nodes but it also affects the runtime decisions made by the system. Well-structured, graph-based application modeling also makes it easier to incorporate autonomic features into each of the application components. Moreover, graph theory algorithms can be exploited during deployment of such an application.

To construct an application graph, we must determine two pieces of information, namely the application components and their interactions, and the resources (i.e. computation time, memory, disk space, network bandwidth) consumed by the components and their interactions. More formally, it is necessary to construct a node-weighted, edge-weighted directed graph $G=(V,E,w_g,c_g)$, where each vertex $v \in V$ represents an application component and the edge $(u,v) \in E$ resembles the communication

from component *u* to component *v*. The computational weight of a vertex *v* is $w_g(v)$ and represents the amount of computation that takes place at component *v.* The communication weight $c_g(u,v)$ captures the amount of communication (volume of data transferred) between components *u* and *v*. Figure 5 illustrates an example application graph with computational and communication weights. When deployed across a distributed heterogeneous environment these weights, along with various system characteristics, such as the processing speed of a resource and the communication latency between resources, determine the actual computation and communication cost.

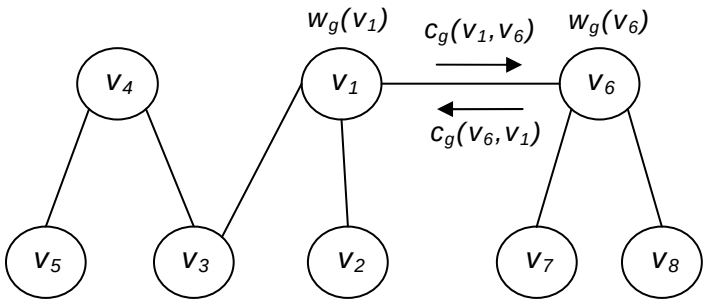


Figure 5. An Application Graph Labeled with Weights.

Static Analysis-based Graph Construction and Resource Inference

Statically analyzing the application code and constructing an application graph signifying the resource requirements of the application components and their links is a challenging task. Exact or even close approximation of these resource requirements requires a great deal of domain knowledge and experience with the specific application, and also involves benchmarking exercises. In our approach, application components are realized as Java objects that encapsulate data, methods and references to other objects.

There are several Java automatic partitioning tools [34,35,37,38], however, they are mostly concerned with determining class-level interactions. Therefore, they perform partitioning at the class granularity and limit the opportunity to exploit object-level concurrency. To our knowledge, Spiegel's Pangaea [36] is the only system that performs analysis at the object level and hence is the only system that could fulfill our purpose. Spiegel's algorithm statically analyzes the Java source code and produces a directed graph, in which nodes and edges represent runtime instances and relations (*create*, *use* and *reference*) among them. We used a modified and extended version of this algorithm to deduce the application graph in this dissertation. In the following sections, we first briefly describe the original algorithm and then discuss the issues related to our implementation.

Spiegel's Original Algorithm

Spiegel defines a set of Java types as classes (non primitive), interfaces and arrays used in the program. He further splits up a Java type as a *static type* (comprising the static fields and methods of the class) and a *dynamic type* (the non-static members). The dynamic part of a Java type is allocated in the heap, whereas the static part is maintained globally on a per-class basis. For each *static type*, precisely one *static object* exists at runtime. For a *dynamic type*, on the other hand, the number of instances depends on the enclosing program statements where the allocation takes place. An allocation may appear inside a control structure or loop, and as a consequence, the number of instances may not be known exactly. In such cases, an *indefinite object* that summarizes all instances is inserted into the object graph. Otherwise *concrete object* nodes are used.

The original algorithm works as follows: initially the set of types comprising the program is computed by syntactically identifying the program's type closure. Then the *Class Relation Graph (CRG)* is constructed that captures relationships at the type level. Whenever a statement belonging to the context of type *A* calls a method or accesses a data field of type *B*, an usage edge $(A,B)_u$ is added to the CRG. Data flow information such as *Export* or *Import* relations take place when new types propagate from one type to another through field access or method calls. An export edge $(A,B,C)_e$ is added between type *A* and *B* when type *A* owns a reference of type *C* and passes it to type *B*. Similarly an import edge $(A,B,C)_i$ is added when type *A* owns a reference of type *B*, from which it receives a reference of type *C*.

The algorithm then computes the *Object Graph (OG)*, which consists of runtime class instances, along with *create* and *reference* relations among them. For each allocation (*new*) statement, *create* and *reference* edges are added between the class instances where the allocation takes place and the newly created instance. After the object population has been computed, the algorithm then iterates over all object triplets and uses the data flow information from CRG to propagate references within OG until a fixed point is reached. Finally, the algorithm adds a *use* edge between objects in OG if they already have a *reference* relation in OG and their corresponding types have a *use* relation in CRG.

Enhancements and Modifications of Spiegel's Algorithm

While Spiegel's algorithm provides important insight about object graph construction, it is not sufficient for our purpose. For instance, the original algorithm

simply produces a directed graph. In contrast, we are interested in a weighted directed graph to effectively extract the computation and communication requirements of the objects and their interactions. Moreover, instead of having a general *use* relation between a pair of objects, our target is to further categorize it as *read-only* and *write* based on whether the data members of an object are simply accessed or modified during use. Such *read-only/write* relations become valuable when configuring them at various distributed sites.

Spiegel's original implementation assumes the presence of source code, while this research performs analysis at the bytecode level. Consequently, we must incorporate significantly different algorithmic aspects and implementation strategies to fulfill the intent of this dissertation. One additional advantage of the bytecode-based approach is that it allows us to access system classes that are accessible at bytecode level. Spiegel's algorithm finds the set of types, objects, methods, etc. by examining the code at the syntactic level. In contrast, our approach uses standard compiler analysis and an efficient Intermediate Representation (IR). Another research [40] also implemented a modified version of Spiegel's algorithm at the bytecode level. However, like Spiegel's approach [36], they also deduced an object graph without computation and communication weights and used a different IR than we did. Determining the type of a runtime object is critical in Java due to polymorphism, inheritance and dynamic binding. Spiegel's algorithm does not use any standard type inference mechanism to resolve dynamic dispatch. Therefore, the set of types that each reference variable may point to at runtime includes all subtypes. Consequently the resultant object graph has unnecessary edges. In contrast, we use

standard techniques such as call graphs and points-to analysis to resolve dynamic dispatch. Consequently, our analysis produces a more compact graph that is less expensive to perform further analysis on. In that way, our approach can manage larger applications.

Our Implementation

Our implementation used the Soot [71] framework for analyzing Java bytecode and is built on top of the Jimple intermediate representation of Soot. The Soot framework is a set of Java APIs for manipulating and optimizing Java bytecode. In order to analyze a complete application using Soot, we first read all class files (including library classes) required by the application. We start with the main method and then recursively load all the class files used in each newly loaded class. As each class is read, it is converted into Jimple IR, suitable for our analysis and transformations. Jimple is a typed, stackless and compact three-address code representation of bytecode. Jimple only involves 19 instruction types. As a result, it is much easier to manipulate compared to stack-oriented bytecode representation, which involves 201 different instructions. The following sections provide the details of the static analysis.

Example Java Application For clarity, we explain the different steps of our implementation by using an example program. Figure 6 shows the example program used throughout this chapter. It should be noted that this example program is not an efficient solution; rather it is deliberately written in a way that helps us describe important concepts of our graph construction algorithm. We show the example in source code for

```
public class Example {
    public static void main(String[] argvs) {
        Student s1 = new Student("James", 76);
        Student s2 = new Student("Jill", 57);
        Course c1 = new Course("CS150");
        c1.addStudent(s1);
        System.out.println(c1.getcName()+":" + c1.findhGpa());
        if(math) {
            Course c2 = new Course("Math100");
            for(int i = 0; i<St_Student.getallStudent().size(); i++) {
                c2.addStudent((Student)St_Student.getallStudent().elementAt(i));
            }
            System.out.println(c2.getcName()+" : " + c2.findhGpa());
        }
    }
}
public class Course {
    private String courseName;
    Vector registeredStudent;
    public Course(String n) {
        courseName = n;
        registeredStudent = new Vector();
    }
   public String getcName() {
        return courseName;
    }
   void addStudent(Student s) {
        registeredStudent.add(s);
    }
   public Student findhGpa() {
        Student hGpaHolder;
        …;
        return hGpaHolder;
    }
}
public class Student {
    private String studentName;
    private double gpa;
    public Student(String name, double gpa) {
        studentName = name;  this.gpa = gpa;
        St_Student.addToallStudent(this);
    }
    public String toString() {
        …
    }
}
public class St_Student {
    static Vector allStudent = new Vector();
    static void addToallStudent(Student s) {
        allStudent.add(s);
    }
    static Vector getallStudent() {
        return allStudent;
    }
}
```

Figure 6. Example Java Source Code.

better readability, although our analysis is performed on compiled Java bytecode. In the example program, there are three classes. Class `Student` describes a student by name and GPA. Class `Course` uses a `java.util.Vector` structure to maintain the list of students registered for the specific course. Class `Example` creates one `Course` object and two `Student` objects and performs operations on them. Another Course object (*c2*) is created only if the user supplied boolean variable *math* is true i.e. when course Math100 becomes compulsory for each student. Class `St_Student` is not part of the original application code and is produced by our system during a preprocessing stage as a result of separating static and dynamic members of original class `Student`.

Call Graph and Points-To-Analysis In this dissertation, Soot's [71] built-in facilities are used to generate a call graph and to perform points-to analysis based on the Jimple representation of the application code. In Java, all instance methods are invoked using virtual calls. The actual method invoked depends on the runtime type of the object receiving the method call and is often termed the receiver object. The call graph approximates the set of target methods that could possibly be invoked at each call site (method invocation) in the program. On the other hand, points-to analysis makes the call graph more compact and precise by limiting the number of target methods invoked in a call site. In particular, Soot's SPARK [72] points-to analysis engine is used to compute the set of runtime types pointed-to by each program variable. Using the set of receiver objects at each call site, for each type, the methods that will be invoked actually are identified.

It is observed that, once the call graph is obtained, it captures all program type interactions if we consider that the only way object *a* of type *A* can access an object *b* of type *B* is by invoking a method defined in type *B*. Identifying interactions that occur whenever an object *a* directly accesses a data field of *b* requires additional passes through the Jimple code and manipulation of some other data structures beside the call graph which is time consuming and introduces additional complexity. Therefore, during preprocessing, additional *accessor* and *mutator* methods are generated at the bytecode level that allow the load and store accesses to data fields via *getXXX()* and *setXXX()* methods, and all direct field accesses in the original program are replaced by respective method call statements. Thus, the analysis is performed at the level of method granularity, which is another major difference from Spiegel's approach. One important additional advantage of this transformation is that not only methods, but fields can be then accessed remotely by a RPC-style mechanism, as implemented in JavaParty. However, this approach cannot be applied to system classes as they are not modifiable.

Method Database This dissertation utilizes an important structure named *Method Database (MD)* to record information about each method, which becomes useful to determine the computational and communicational weights. The MD is constructed by inspecting the Jimple representation of each method used in the program. Each such method is analyzed to obtain information such as 1) whether it involves reading or writing of the fields of the class containing the method 2) the amount of resource (CPU, memory) consumed by the method and 3) the amount of communication needed to

invoke the method. To avoid ambiguity due to polymorphism and method overloading, each method in the MD is represented by a unique method signature as follows:

<className: retType methodName(parType$_1$, parType$_2$ … … … parType$_n$)>

Each method is categorized as read/write by recursively examining the load/store field accesses in the method's context. Approximating the computation time for each method in Java is a significant challenge. Moreover, the presence of conditional statements, loops and dependability on both input parameters and hardware platforms make it more difficult to estimate the execution time. A recent study [73] suggests that bytecode instruction counting (BIC) can be used as a portable CPU consumption metric instead of traditional clock elapsed time. Motivated by this study, in this dissertation the computational weight of each method is estimated in terms of the number of Jimple instructions needs to be executed (*ins_cost*). We believe that the relationship between the total number of Jimple instructions and the total number of bytecode instructions needed to execute a particular method is almost linear. Hence, the former can be used as a replacement.

Figure 7 shows a Java method and its Jimple representation, where categorization of each Jimple statement is also included. Each of the 19 types of Jimple instruction is then weighted according to the associated cost and the final estimate for a method is generated based on the weighted sum. In the case of control structures, it is assumed that each branch of an if-else is taken 50% of the time and loops are executed for a configurable number of times [74]. The architecture-specific execution costs associated with *ins_cost* can then be computed by *arch_cost,* provided in units of *msec* per Jimple

instruction. To obtain the *arch_cost*, the Jimple code is benchmarked on each type of resource. The computation time obtained in this way is approximated, and parameters can be tuned over time to provide a more accurate estimate. The memory requirement of each method is relatively straightforward to model. The memory consumed by a dynamic object is calculated as the sum of the memory required to store its data members, method parameters and local variables. For a static object, the required memory is estimated as the sum of its code and static data size.

```
public void test(int x) {
    double sum;
    if(x<10) {
        String s = "Less than ten";
        s.toString();
    }
    else sum = Math.sqrt(x);
}

public void test(int) {
        Test r0;
        int i0;
        java.lang.String r1;
        double $d1;
        r0 := @this: Test;              // identityStmt
        i0 := @parameter0: int;         // identityStmt
        if i0 >= 10 goto label0;        // ifStmt
        r1 = "Less than ten";           // assignStmt
        virtualinvoke r1.<java.lang.String: java.lang.String toString()>();
                                        // virtual method invocation
        goto label1;                    // gotoStmt

    label0: $d1 = (double) i0;          // assignStmt
            staticinvoke <java.lang.Math: double sqrt(double)>($d1);
                                        // static method invocation
    label1: return;                     // returnStmt
}
```

Figure 7. Java Method and Respective Jimple Code.

Finding the Set of Types To find the set of types of an application, at first the corresponding call graph is built using the Soot framework. The set of types of an

application is exactly the set of types that appears as the receiver of the methods in the call graph.

Construction of the Class Relation Graph (CRG) The CRG is a directed graph $G_c=(V_c,E_c)$ where V_c is the set of types that we computed in the previous step and E_c is the set of *use*, *export* or *import* edges among types as explained in the original algorithm. The CRG construction algorithm explores every node (method) in the call graph and works as follows:

For each method invocation

1) Let *A* be the class containing the method call statement, *B* be the receiver class of the method, *P* be the list of types of method parameters and *R* be the return type.
2) If *A* != *B*, then add the following edges between type *A* and *B*,
   a) An use edge $(A,B)_u$
   b) An export edge $(A,B,P[])_e$
   c) An import edge $(A,B,R)_i$

Figure 8 shows the CRG deduced from the example program. For better visualization, we summarize all usage, export and import edges between type *A* and type *B* into a single edge as *(A,B,E[],I[])*, where list *E[]* contains a set of types exported from type *A* to *B* and list *I[]* contains a set of types imported from type *B* to *A*. It should be noted that the Java runtime system classes that are also invoked by the references to classes `System.out` and `String` are omitted in the final CRG.

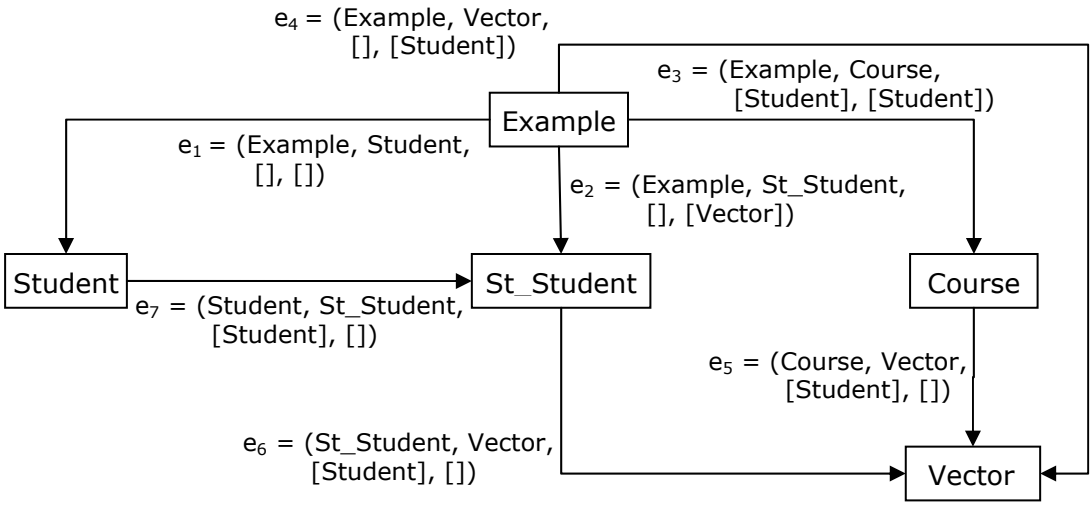


Figure 8. The CRG of the Example Program.

Each edge *(A,B)* in CRG also contains a list of methods through which type *A* and *B* interact. With this list and information contained about each method in MD, the edge *(A,B)* in the CRG is then further categorized as *read-only* or *write.* Moreover, when the directed edge *(A,B)* in CRG contains the method *M* in its methods list, the computation weight of type B, *w(B)* is specified as the computation weight and memory usage of *M* stored in MD. The communication weight *c(A,B)* depends both on how many times *A*'s enclosing statements invokes a method of *B* and the number of bytes required to represent the method parameters and return types. If the method parameter is an object, the size of the whole object is considered while calculating the communication weight. Final values of *w(B)* and *c(A,B)* are calculated by adding the weights generated by all methods appearing in edge *(A,B)*. For example edge *(Example,Course),* which appears in Figure 8, involves following four methods:

```
Course: void <init>(java.lang.String)
Course: void addStudent(Student)
Course: java.lang.String getCourseName()
```

```
Course: Student findhGpa()
```

Some of these methods modify data members of class `Course`, so the edge *(Example, Course)* is categorized as *write*. The weight *w(v)*, where *v* is an object of type `Course`, is then the summation of computations performed by the above methods.

Construction of the Initial Object Graph (OG) The OG is a weighted directed graph *G=(V,E)* where *V* is the set of runtime instances and can be of type *static*, *concrete* or *indefinite* as explained previously and *E* is the set of edges among the nodes in *V* and can be of type *read-only, write*, *create* or *reference*. To obtain the set of objects, the allocations in the program are identified. In Jimple, this includes statements that allocate objects and arrays and that load string constants. Some examples of the allocation statements in Jimple are,

```
p = new Student,  q = newarray(int)[12], r = "Hello"
```

The allocation statements are identified at the same time as the method database is constructed, as both require examination of each Jimple method. Each method in MD keeps track of all allocations inside it, including the class of the allocated object and the type of allocation (*concrete* or *indefinite*). Static edges are also identified in CRG, where the receiver object of the method call is of type *static*. For instance, edges *(Example,St_Student)* and *(Student,St_Student)* in Figure 8 are static edges as identified in Jimple code, where static invocation is always preceded by `staticinvoke` such as in Figure 7. Figure 9 shows the object graph generated from our example program as constructed by the following algorithm:

1) For each static type, add one static object in the OG.
2) Repeat the following for all static objects in the OG
   a) For a static object of type *A*, extract all the allocations from all the methods in the MD where the *className* part of the method signature matches *A*.
      i) Add one concrete object of the respective type for each concrete allocation in *A*.
      ii) Add one indefinite object of the respective type for all allocations of that particular type inside a control dependent statement in *A*.
   b) For each concrete object of type *A* added in the graph, add concrete/indefinite objects in the same way as in step 2(a).
   c) For each indefinite object of type *A* added in the graph, all allocation statements in *A* are treated as uncertain and therefore an indefinite object is added to the graph for all such allocations in *A*.
3) For each concrete or indefinite object added to the graph, add a create and a reference edge from the parent to the newly created object.
4) For each static edge *(A,B)*, add a reference edge from all objects of type *A* to static object *B*.

Propagating Use and Reference Edges The algorithm now iterates over all triples of objects *(a,b,c)* in the OG for which *reference* edges $(a,b)_r$ and $(a,c)_r$ or $(b,c)_r$ exist in the OG and matches the types of the objects against the data flow edges $(A,B,C)_e$ or $(A,B,C)_i$ in the CRG. Then a new *reference* edge $(b,c)_r$ or $(a,c)_r$ is added to the object

graph, as explained in Figure 10. Finally a *use* edge $(a,b)_u$ is added to the OG if (A,B) exists in the CRG and $(a,b)_r$ exists in the OG. For each added *use* edge, the corresponding types of usage and computational and communicational weights are also propagated from the CRG to the OG. For clarity, the object graph in Figure 9 only shows *read* and *write* relations between objects while omitting the creation and reference relations. It also excludes JDK library objects except for the dynamic `Vector` object.

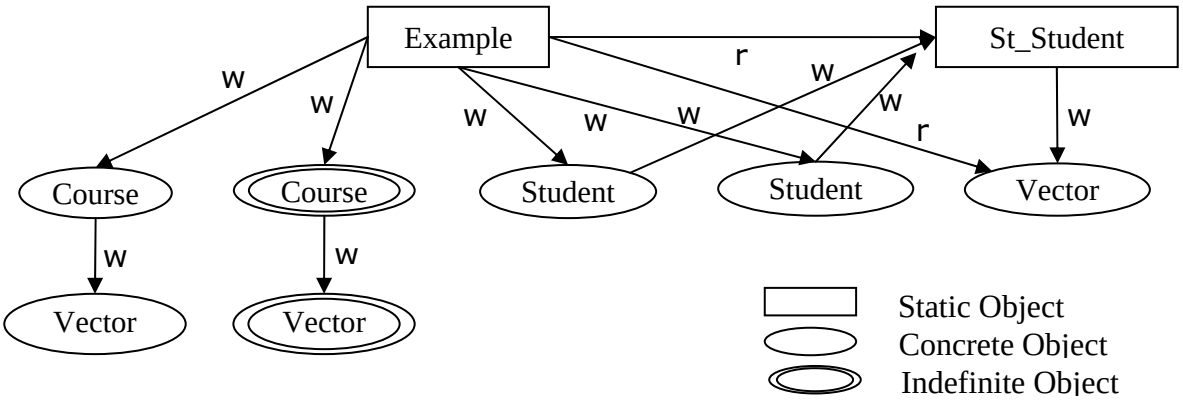


Figure 9. The OG of the Example Program.

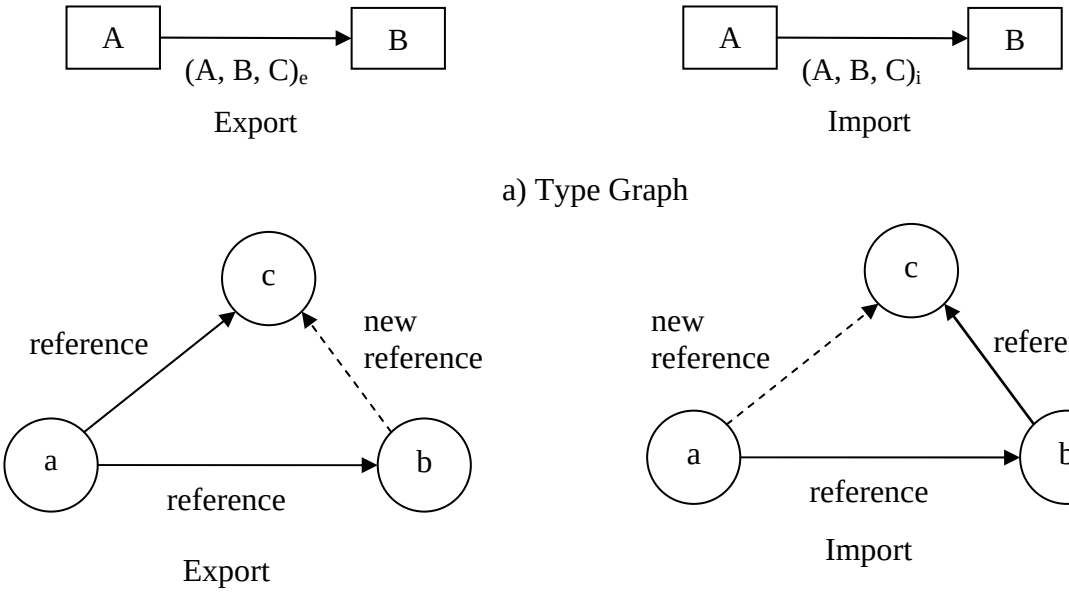


Figure 10. Data Propagation from CRG to OG.

Indefinite Objects In master-worker applications, there is often an indefinite number of `Thread` objects, usually created in loops controlled by the configuration parameters i.e. their actual number is not known statically. The resulting `Thread` objects, and all further objects created by them, will therefore be represented as *indefinite* objects in the object graph. The situation is explained in the example program. When in class `Example` an object of type Course is created inside a conditional statement. Hence, if we assume the *indefinite* object representing all `Thread` objects as a single object, then all the worker threads would be allocated to a single node for execution. This problem is resolved by specially treating `Thread` objects and deferring the decision of how many worker objects the indefinite object actually represents until runtime. During execution, the distribution then becomes straightforward by placing the master object at the root node, where the application initiated, and allocating *n* objects of type `Thread` to other available nodes according to some appropriate strategy. Every other object created by a specific `Thread` object is assigned to the same node as its parent.

Distribution-relevant Properties Based on the object graph, it is possible to categorize the objects as 1) immutable (has no incoming *write* edge, i.e. never changes after being created), 2) Single Read/Write (Object *A* is only accessed by object *B* and no other object), 3) Single Write Multiple Read (Object *A* is written by object *B* but read by many others) and 4) Multiple Read Multiple Write (Object *A* is accessed by many others). Effective distribution decisions can then be generated for each object category, such as: replicate objects of category 1) across the system, objects of category 2) do not

need to be remotely invokable, they only need to be co-located with their accessor object, objects of category 3) need to be remotely invokable and initially co-located with their writer objects etc. The objective is to place the most communicating objects on the same machine, thereby minimizing network traffic. To automatically distribute a wide variety of object-oriented applications, such classification and distribution policies for an object, along with its computation and communication weights, play a significant role during its initial configuration and later rearrangement.

Distributed Code Generation Once the objects and their interactions are extracted, distributed code is generated to transform the program into a distributed version through the use of bytecode rewriting. For remotely accessible objects, proxy objects pass the references indirectly to the original objects. Therefore, proxy classes are generated and classes that allocate or refer to remote objects need to be modified. The transformed program can then be executed in the targeted self-managing distributed environment, which handles allocation and invocation requests to remote objects. Figure 11 shows the distributed execution for some of the objects of the example program with proxy objects. However, the actual assignment of the objects in the distributed platform is performed in the autonomic layer of the system architecture and Chapter 6 of this dissertation details the deployment strategies.

Limitations of Static Analysis

In the form presented here, our static analysis-based application graph construction algorithm performed well to understand the program structure and resource requirements

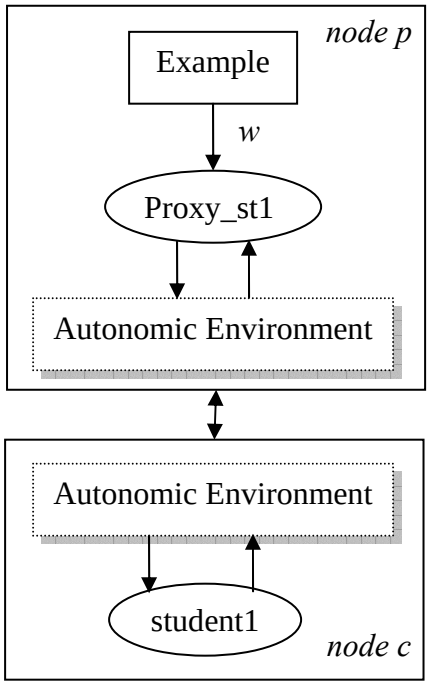


Figure 11. Distributed Execution with Proxy Objects.

as demonstrated by the example program. However, there is one significant limitation of this approach that has an effect on its practical applicability. As our objective is to perform a correct and complete analysis to extract the most accurate estimation of the runtime structure of an application, our analysis not only covers the application classes but also includes the vast Java library classes. The resultant object graph size is therefore considerable, which makes it difficult to perform further analysis to extract the object deployment strategies. The numbers of *reference*, *read* and *write* edges are also very high, and may be problematic for larger programs. Therefore, in order to extract useful graph abstractions for larger applications it is necessary to refine the object graph generated by the static analysis approach. Currently, this is out of the scope of this dissertation, but we would like to pursue it in the future.

CHAPTER 5

THE NETWORK LAYER

This chapter presents the utility-aware tree model that represents the underlying computing environment. The chapter first justifies the choice of a tree to model the network and discusses its suitability to represent the computing environment considered in this dissertation. Then it provides the formal definition of the tree abstraction and elaborates the overlay tree network construction process. Finally, the chapter presents the resource monitoring module utilized in this dissertation for tracking distributed resources.

Tree Representation of the Network

In this research, the target environment for the deployment of the application is a distributed environment consisting of a non-dedicated heterogeneous and distributed collection of nodes connected by a network. A resource (node) in this environment could be a single PC, laptop, server or a cluster of workstations. Therefore, each node has different resource characteristics. The communication layer that connects these diverse resources is also heterogeneous considering the network topology, communication latency and bandwidth.

To organize the computation around this heterogeneous and distributed pool of resources, traditional approaches [75,76] rely on the assumption that sufficiently detailed and up-to-date knowledge of the underlying resources is available to a central entity. While this approach results in the optimized utilization of the resources, it does not scale

to a large numbers of nodes, nor does it deal with real-time changes in the underlying infrastructure. Maintaining a global view of a large-scale distributed environment becomes prohibitively expensive, even impossible at a certain stage, considering the unprecedented number of nodes and the unpredictability associated with a large-scale computing system due to various dynamic factors.

We propose a different approach that addresses the above problems and allows the heterogeneous pool of resources to be organized in a structure that facilitates their effective use. The aim is to organize the distributed resources in a structure such that nodes that are *closer* to each other in the structure are also *closer* to each other considering network distance (latency, bandwidth, etc.). Once structured in this way, it is possible to detect higher utility paths locally that correspond to low latency and high bandwidth between network nodes. As a result of that, the deployment of the application graph can be performed in a utility-aware way, without having full knowledge of the underlying resources and without calculating the utility between all pairs of network nodes.

The abovementioned organization is obtained by modeling the target distributed environment as a tree in which the nodes correspond to compute resources, edges correspond to network connections and execution starts at the root. More specifically, a tree structured overlay network [77, 78] is used to model the underlying resources, which is built on the fly on top of the existing network topology. Such a tree overlay is utilized in several studies [77, 78, 79], where the tree structured computing environment is deployed for master-worker type computation. Among these studies, only one [78]

addresses the dynamic adaptation in a limited way. On the contrary, this dissertation utilizes the tree organization to deploy our targeted applications and to realize their dynamic reconfigurations.

Having such a tree-shaped computing environment, a simple but effective autonomic deployment algorithm (details are provided in Chapter 6) is used to organize computation on the available nodes. Intuitively, each node in the tree decides which components to execute by itself and which ones to delegate to its children. Delegation is performed in a utility-aware manner, meaning that each parent monitors its children and based on that information selects its best child subtree for delegating certain application components.

There are three main advantages to the tree model and the associated autonomic scheduling algorithm. First, the approach is scalable and adaptive. In this approach, by limiting the number of children, the amount of information needing to be maintained at each parent can be bounded. Therefore, each node monitors only a certain number of other nodes (its children) and delegates computation to these nodes based on its interaction with them and with the environment. Each deployment decision is made based on minimal knowledge available locally, which may not be optimal considering centralized deployment, but scales to a large number of nodes. Each node also can change its scheduling to adapt to the changes in the environment (e.g. the computing resource becomes overloaded or inaccessible, communication becomes slower due to congestion, etc.). Also the tree model can grow and reconfigure itself to accommodate a larger application and to adapt to the dynamically evolving computing environment.

Second, this model relieves us from the costly evaluation of the utility function globally by limiting the utility evaluation within a subtree performed by the parent of that subtree. Each parent is capable of monitoring its children and calculating the corresponding utilities. As a result of that, the parent is capable of reconfiguring its subtree, based on the changes in utilities of its children. Optimizing certain utility functions globally is certainly more attractive, however to achieve that it is necessary to know the computational and communication capabilities of all the resources in the network. The approach therefore does not scale very well when the number of deployed applications and/or number of resources grow in the system. On the other hand, this research may not be able to optimize utility or resource allocation, but by reducing the problem of evaluating and maintaining the utility across the whole system to the problem of managing the utility within a sub-tree, it promises to provide better adaptability and scalability in such a dynamic environment.

Third, the model fits very well with the classes of applications that interest us. In the case of master worker type applications, the master component, originating at the root node of the tree, decides which worker tasks to execute, and which others to delegate to its children. In turn, each child decides which tasks to execute and which others to forward to its own children. In inherently distributed applications, the applications components are divided among partitions and the partitions are then propagated down the hierarchy. Even in the case of applications containing a large number of communicating components, there is still a single entity that initiates the set of communicating

components and allocates them to processors. It is natural to think of the initiator as the root of the tree.

Formally, the entire network is represented as a node-weighted link-weighted tree $T=(N,L,w_t,c_t)$, where N represents the set of computational nodes and L represents network links among them. Each node $n \in N$ represents a computing resource of weight $w_t(n)$, meaning that the node n requires $w_t(n)$ time for each unit of computation. Each edge $(m,n) \in L$ corresponds to a communication link weighted by $c_t(m,n)$, which represents the time needed by a parent node m to communicate one task to its child n considering both bandwidth and latency. All weights are assumed to be positive rational numbers. When two nodes are not connected directly, their communication weight is the sum of the link weights on the path via their predecessors or successors. Therefore, larger values of node and edge weights translate to slower nodes and slower communication respectively. Figure 12 shows a small computing environment where resources are distributed in three domains. Figure 13 illustrates a tree overlay network that is built on top of the physical network topology existing among the nodes in the computing environment. The detailed process of overlay tree construction is discussed in the next section.

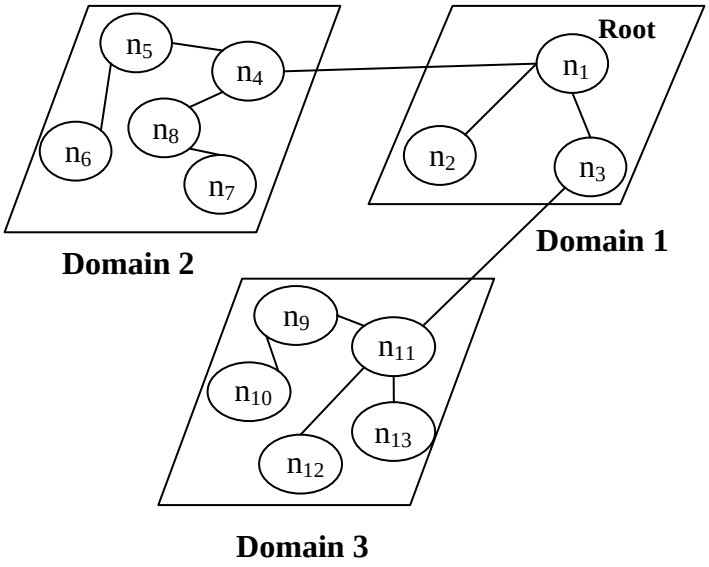


Figure 12. Sample Distributed Environment.

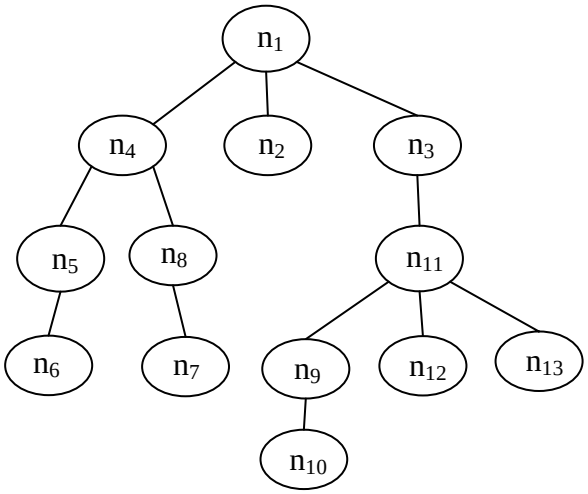


Figure 13. Overlay Tree.

Overlay Tree Construction

An overlay tree network does not exist at the beginning of the computation, it has to be built on top of the existing topology. To construct an overlay tree, each node is assumed to have a *children list* signifying the URLs of its neighbors that have direct

connections with it. The problem of how to generate this list is out of the scope of this research, however it can be addressed by using techniques used in tools such Gnutella [80] or CAN [81]. For our purpose, we are assuming the presence of an initial list at each node that is utilized by the autonomic element managing that particular node to propagate computations allocated to it. Once a user starts an application in his/her machine, the graph representation is extracted from the application code. The initiator node then decides which components to execute and which ones to delegate to the its best utility child nodes considering all the nodes listed in its children list. The delegated nodes again spread the computation in this manner. The topology of the resulting overlay network thus becomes a tree with the originating machine at the root node. The next sections detail important aspects of overlay tree construction process.

Children List

In our design, a node can not have an arbitrarily large number of children because it will be costly to maintain an up-to-date view of a large number of children in a dynamically changing environment necessary for utility calculation. Moreover, if there are significant communications between the parent and its children, the communication bandwidth of the parent becomes a limiting factor that determines the number of nodes that can be successfully communicated with from the parent. On the other hand having too few nodes as children may lead to a deep tree, which consequently will incur delays while propagating the computation from the root to the leaves of the tree. Therefore, in our design, each node has a set number of children, *c,* chosen by through experimentation. The initial list may contain more than *c* children, but only the best *c*

children remain *active* at a certain time. A parent node monitors every child that is part of its initial list, and based on that data some previously *active* children may become *dormant* (because of their poor performance due to workload, congestion, etc.) and some *dormant* children may become *active*.

Broken Link and Cycles in the Tree

In our design presented here, a child is only accessible through its parent. However, if the parent becomes inaccessible due to processor or network failure, the node and its children become unreachable. Tasks allocated to that parent and subsequent children become useless. To avoid this situation, a child must be able to find a predecessor, if necessary, other than its own parent. In our design, as the computation propagates, every parent adds itself to a *predecessor list* and passes this list to its children along with the subtasks. Due to reconfiguration or self-optimization, the topology of the overlay network may change at runtime and whenever this happens an updated predecessor list is forwarded to each child. If a child is unable to get a response back from its parent after waiting a certain amount of time, it starts sending messages to the nodes listed in its predecessor list one by one. Whenever the child receives a reply back from an alive predecessor, it becomes the new parent of the child and sends the child back a new predecessor list. If none of the predecessors reply, the child then terminates its execution. Failed leaf nodes are detected by corresponding parents during usual monitoring. If a leaf node fails, the parent node just discards the tasks delegated to that node and reallocates it to some other node.

The initial overlay network is cycle free, as the assumption is that the initial children list at each node is created in a way that eliminates cycles. However, due to failure and reconfiguration, cycles may form during the execution. To prevent this from happening, each node, whenever asked to handle some computation by its parent, examines the predecessor list obtained from its parent. If the node detects itself in this list, it knows that it is already handling computation at some level of the tree and discards its parent's request for further delegation.

Resource Monitoring

Each parent periodically monitors the nodes in its *children list.* Each node corresponds to an instance $r = \{a_1, a_2 \ldots a_j\}$, where each a_i measures the value of some performance attribute of that node. The performance attributes are measurable entities involving topological, computational and communication aspects of a particular node. Based on this information, the parent calculates the utility achieved by assigning subtasks to a certain child and selects the best-utility child for delegation.

CHAPTER 6

THE AUTONOMIC LAYER

This chapter presents the algorithms and techniques that have been designed and implemented to achieve self-configuration and self-optimization of the deployed application graph. The first section discusses the formulation of the utility function that drives the initial deployment and further optimization in this research. The section also discusses the high-level policies adopted in this research and their incorporations into the utility function. Finally the section elaborates how the proposed utility function can be formulated and parameterized to achieve application specific utility. The next section presents the autonomic deployment algorithm that finds the initial configuration of the application components to the network nodes. This section starts by having an intuitive description of the initial deployment algorithm, it then explains the utility calculation in detail, finally it presents the algorithm formally. The final section of this chapter presents the techniques devised to initiate reconfigurations towards optimizations. Works presented in this chapter appears in the following publications: [82, 83].

Utility Function

The efficiency of a distributed system can be measured in many different ways such as by minimizing application execution time, by maximizing throughput, by maximizing resource utilization, by satisfying certain user defined policies or by maximizing economic benefits. Typical approaches try to achieve these high level

objectives one at a time and employ rule-based or goal-based policies or an objective function that optimizes a single factor. This dissertation proposes a different approach that combines several applications and environment specific attributes in a single utility function. This multi-attribute function returns a scalar value signifying the system's overall utility for each possible state of a system and the goal becomes to select a state that maximizes the system's overall utility. As computing environments are becoming increasingly large, distributed, complex and dynamic in nature, the optimal actions are likely to evolve over time and a utility function that continuously computes the most desired state is expected to be more suitable in such cases.

In our approach, the utility functions are application specific and can be customized according to the respective application's high-level objectives. As discussed in Chapter 1, our target application classes have various requirements. For some of them, it may be important to minimize the total execution time, for some others, it may be more important to attain low latency, high throughput data transfer than a quick response time. Additional environment and user specific policies considered in this dissertation are to maximize resource utilization, to prefer higher priority jobs, to adapt to network conditions, etc. Having all these issues, we have designed the utility function to respect the following application, environment and user specific high-level policies:

1. While mapping partitions containing a large number of application components in the tree network, nodes that lead to a wider subtree (higher degree of connectivity) are preferred as a higher degree allows more directions for partition growth.

2. Faster and less busy nodes are favored over slower and overloaded nodes when assigning components to resources.
3. Nodes with faster communication links are preferred over nodes with slower communication links.
4. High priority applications are preferred over low priority jobs.

In order to derive a suitable utility function, various attributes such as processor speed and load, network latency and bandwidth, job priority, node connectivity, etc. are identified for incorporation into the utility calculation. With these attributes, the template of a simple but effective utility function is then composed that calculates the overall system utility based on these metrics and also works in accordance with the abovementioned policies. The template is defined as follows:

$$U = \frac{p \times d}{c_p + c_m} \qquad (Equation\ 1)$$

where the parameters p, d, c_p and c_m indicate the application priority, node connectivity, computational and communication cost respectively. With this template, a user/administrator can customize an appropriate utility function that expresses application specific requirements by incorporating suitable values/functions for these parameters. Automatic inference of the parameter values could be achieved by using certain techniques such as feedback from ongoing computations, extensive offline measurements, or machine learning. Currently these approaches are out of the scope of this dissertation; however we would like to explore them in the future.

As an illustration, following is an example utility function based on the above template (Equation 1) that puts more weight on communication (the utility decreases

quickly with increasing communication cost) and therefore is suitable for communication/data intensive applications.

$$U = \frac{\delta(p \times d)}{(\alpha c_p + \beta) + \frac{c_m{}^2}{\gamma}}$$

where α, β, γ, and δ are constants. Figure 14 plots the corresponding utility against each of these four parameters independently, keeping the others constant. The computation and communication costs are varied randomly in the range 10-100. Application priority and node connectivity are varied from 1 to 10, but expressed as multiplied by 10.

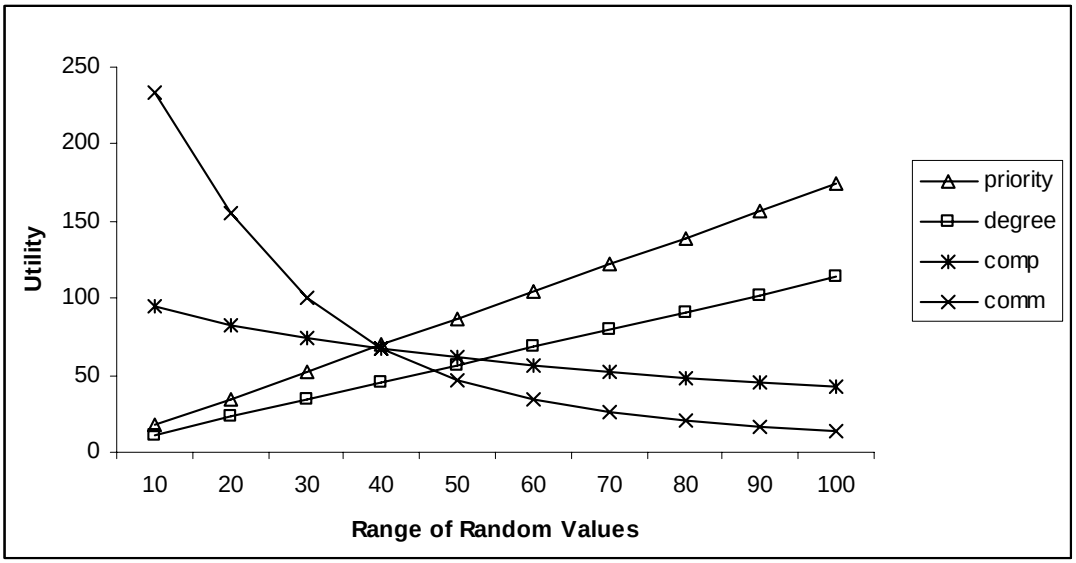


Figure 14 . An Example Utility Model for Four Individual Parameters.

The plots demonstrate that utility has a positive correlation with the application priority and node connectivity, while other parameters are held constant. On the other hand, utility decreases with the increase in communicational and computation cost. As required for communication/data intensive applications, the most rapid decline in utility

occurs with increasing communication, whereas the effect of increasing computation cost is minimal. In particular, the example utility function demonstrates the importance of selecting higher utility network links that minimize communication overhead.

Initial Deployment

Once the application and underlying resources have both been modeled, the deployment problem reduces to the mapping of different application components and their interconnections to different nodes in the target environment and network links among them so that all requirements and constraints are satisfied and the system's overall utility is maximized. The assumption is that the application can be submitted to any node, which acts as the root or starting point of the application. Furthermore the application may end its execution either at the root node or at one or more clients at different destination nodes. The nodes in the system are structured into a tree-shaped environment rooted at the source of execution according to the model described in Chapter 5. An intuitive description of the autonomic deployment algorithm that is utilized in this dissertation to organize computation on the available nodes is presented below:

> Each node in the tree either completely executes the tasks assigned to it or divides the computation (if it is too large to execute by itself within a reasonable amount of time) and propagates the parts down the hierarchy. Each parent node is capable of calculating the utilities of its children and during each delegation selects its best child's subtree to forward a particular task to considering all the children available at that time for delegation.

Once the application graph G is submitted to the root node of the tree network, the root decides which application components to execute itself and which components to forward to its child's sub-tree. The child, who has been delegated a set of components again deploys them in the same way to its subtrees. For effective delegation of components at a particular node having $|P|$ children, graph coarsening techniques [84] are exploited to collapse several application components into a single partition, so that $\leq |P|$ partitions are generated at that stage. The coarsened graph is projected back to the original or to a more refined graph once it is delegated to a child node.

In the above approach, each parent selects the highest utility child to delegate a particular partition (set of components). Finding the highest utility child to delegate a partition to means finding the highest utility mapping M of the edges (v_j,v_k) where $v_j \in V_r$ (represents the set of components that the parent decided to execute itself) and $v_k \in V_s$ (represents the set of components that belong to a partition that a parent decided to delegate). More formally, a mapping needs to be produced, which assigns each $v_k \in V_s$ to a $n_q \in N$ in a way such that the network node n_q is capable of fulfilling the requirements and constraints of application node v_k and the edge (v_j,v_k) is mapped to the highest utility link considering all children available at that stage for delegation. The utility of an edge (v_j,v_k) is represented as $U(v_j,v_k)$, and returns the utility achieved due to the mapping of the edge (v_j,v_k) on certain network link. More specifically, based on the template of the utility function (Equation 1), the utility of an edge (v_j,v_k), while mapped to the network link (n_p,n_q), where n_p represents the parent in the tree-shaped network where v_j is already

mapped and n_q represents a potential child for delegating application component v_k, is calculated by using the following function:

$$U(v_j, v_k) = \frac{d(n_q)}{f_1(w_g(v_k) \times w_t(n_q)) + f_2(w_g(v_j, v_k) \times w_t(n_p, n_q))} \quad (Equation\ 2)$$

where $d(n_q)$ represents the number of children of network tree node n_q, function f_1 models the cost of processing vertex v_k in node n_q and f_2 models the cost resulting from mapping edge (v_j,v_k) to link (n_p,n_q). Functions f_1 and f_2 must be defined carefully to satisfy the application specific utility.

The utility model in the above scenario is the "highest-degree child with the fastest computation capability and fastest communication link". To ensure that the application graph partitions with the largest number of components are delegated to the highest degree child, candidate partitions are sorted according to their sizes and then deployed according to that order. In the case of simultaneous scheduling of multiple applications with different priorities, the system needs to guarantee that higher priority applications execute before applications with lower priority. To achieve this, applications are ordered according to their priorities and then mapped following that order. The overall utility of an application graph G with priority p due to deployment M is then calculated as:

$$U(G, M) = p \times \sum_{(v_j, v_k) \in E} U(v_j, v_k) \quad (Equation\ 3)$$

Therefore, at the level of an individual application the problem of self-configuration becomes the problem of finding the highest utility mapping M between edges E in the

application graph and the links L of the network graph. Formally, the self-configuration algorithm works as follows:

1) The application graph $G=(V,E)$ is submitted as input to an arbitrary node $n_p \in N$, which becomes the root for execution.
2) For each edge $(v_j, v_k) \in E$, where v_j represents the partitions already mapped to the node n_p, the highest utility mapping is found that maps the edge (v_j, v_k) to the network link $(n_p, n_q) \in L$ according to the abovementioned utility calculation.
3) This mapping partitions the graph G into a number of sub-graphs each allocated to a different child of the node n_p.
4) The sub-graphs are then again deployed similarly in the corresponding child's subtree. The process continues until there are no more components available for delegation.

Self Optimization

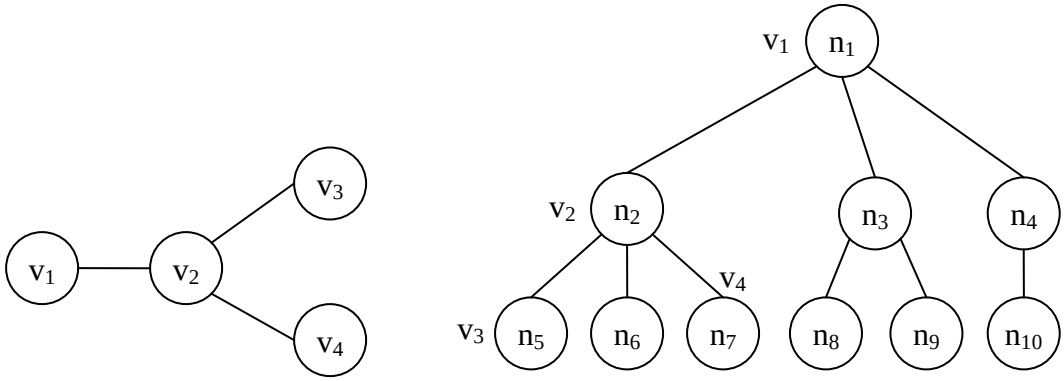
After initial placement, the environment may change and as a result the utility may drop. Therefore, it is necessary to monitor the utility and trigger reconfiguration as required. In this dissertation, reconfiguration is triggered in response to a variety of events such as changes in network delays, changes in available bandwidth, changes in available processing capability, etc. Some business specific events may also trigger reconfiguration such as the arrival of a higher priority job, etc.

Reconfiguration is costly and disruptive; therefore, it is not feasible to initiate reconfiguration unless it is productive. In our design, reconfiguration is performed only within a subtree and therefore is expected to be a less expensive process because of the

way the underlying network is modeled. More specifically, reconfiguration is performed on an individual edge basis i.e. when the utility of a certain application graph edge drops below a certain threshold (user specified or system generated by comparing the utility during initial deployment), that edge is reconfigured to another network link. In that way, reconfiguration is kept to a minimum by allowing a parent to discard a problematic link/child and move the computation to another child.

The abovementioned edge-based reconfiguration works well for the graph edges that connect vertices where the execution ends, as in that case redeploying that end vertex to a different network node does not have any impact on the other ongoing computations in the other parts of the tree. However, the approach becomes inefficient for the edges that connect intermediate application graph vertices. We explain the dilemma and our solution with an example scenario as illustrated in Figure 15, which shows an application graph structure and the initial configuration of the graph vertices to the network tree nodes. It is straightforward to find the reconfiguration of edge (v_2,v_4) once the computation or communication speed of the network node n_7 changes. However, while redeploying edge (v_1,v_2), it is not sufficient to choose the best utility child, e.g. n_3 for mapping the vertex v_2. As vertex v_2 has interactions with vertices v_3 and v_4, the reconfiguration also needs to minimize the communication cost between the network node, where v_2 will be mapped after reconfiguration, and the network nodes where vertices v_3 and v_4 are mapped currently. Therefore, whenever the decision is made to map v_2 to n_3 rather than n_4, it means that considering both the utility and the cost needed to communicate other nodes mapping other components connected to v_2, n_3 appears to be

more desirable than n_4. Once v_2 is reconfigured to say n_3, it becomes the new parent of the nodes n_5 and n_7, where vertices v_3 and v_4 are placed.



a) Application Graph. b) Initial Configuration.

Figure 15. Explaining Reconfiguration Techniques.

Through the resource monitoring module, each parent node periodically measures the workload at each child and its bandwidth to the child and consequently changes the computational and communication weights of that child. By incorporating this monitored information into the utility function, the parent observes the change in the utility due to the changes in the network and the compute nodes. Reconfiguration of the application graph edge (v_j,v_k) while mapped to the network link (n_p,n_q) is initiated autonomously when the following condition satisfies.

$$U\left(v_j, v_k\right)_b - U\left(v_j, v_k\right)_c \geq \varepsilon$$

In the above equation, $U(v_j,v_k)_c$ is the utility currently being achieved with the current configuration and $U(v_j,v_k)_b$ is the utility that is achievable with a better configuration and ε is the threshold value. As parent n_p calculates the utilities of its children on a recurrent basis, it keeps tracks of the current utility $U(v_j,v_k)_c$ achieved by mapping the application component v_k to the network node n_q. The parent also seeks better utility that may result

by mapping the component v_k to some other child node available for delegation. In case, there is a utility significantly better than the current utility, the reconfiguration initiates. Due to the dynamic communication inside the network, configurations that offer significantly better utilities are expected to emerge and application components are expected to reconfigure accordingly to optimize the execution.

CHAPTER 7

EXPERIMENTS AND MEASUREMENTS

In this chapter, we evaluate the performance of the different aspects of the self-managed deployment using a simulation study. The experiments were performed in a dual, quad-core Xeon processor with 16GB of RAM. We used the system's overall utility as the performance metric in all our experiments. This chapter first describes the simulation setup, test applications and utility function used for the experimentations. It then presents the experiments and discusses the results. Results presented in this chapter also appears in the following paper: [85].

Simulation Setup

One of our main objectives is to perform the study and evaluate our algorithms in an actual large-scale, shared, heterogeneous, and highly-dynamic distributed environment. A network of this kind is rarely available for experimental purposes because of the associated cost, security and management complexity. Instead of evaluating our approach in a specific and trivial network setting, composed of a limited number of nodes connected by a particular topology that we have access to now, we assessed our solutions using simulation.

Several topology generation tools are available for creating a desired model of the network [86, 87, 88, 89]. We used the GT-ITM internetwork topology generator [89] to generate a sample large-scale, heterogeneous computing environment for evaluating our

self-configuration and self-optimization techniques. We have chosen this topology generator because of its Transit-Stub model, which correlates well with the structure of the Internet, including hierarchy and locality. Also, the topology generated by the GT-ITM is easy to incorporate into the ns-2 network simulator [90], which was used to simulate communication inside the topology. Similar to today's Internet, in a Transit-Stub model, transit domains represent the backbone topology, where nodes are connected via high-bandwidth links. Each backbone node in the transit domain connects to a gateway node in a stub domain, which is essentially like a LAN connecting a number of nodes. One stub domain communicates with another via high-connection nodes in the transit domain. For our study, we used a topology with one transit domain containing four transit nodes; each connected to the gateway of a stub domain representing 32 nodes. Therefore, the total number of nodes used in our study was 4+4x32=132. The topology also contained 993 duplex connections among nodes. GT-ITM also generates a propagation delay for each network link proportional to its length. To reflect the processor heterogeneity, GT-ITM generates processing weights that vary randomly across a wide range. Figure 16 elaborates the experimental structure of the network topology used in this dissertation.

To generate traffic that simulates real world workload and bandwidth consumption in a shared environment, we used the traffic generator available in the ns-2 simulation package. The traffic generator script cbrgen.tcl is available under the ns-2 package to create CBR traffic connections between network nodes. The network topology generated by the GT-ITM is transformed to the equivalent ns-2 topology by using a

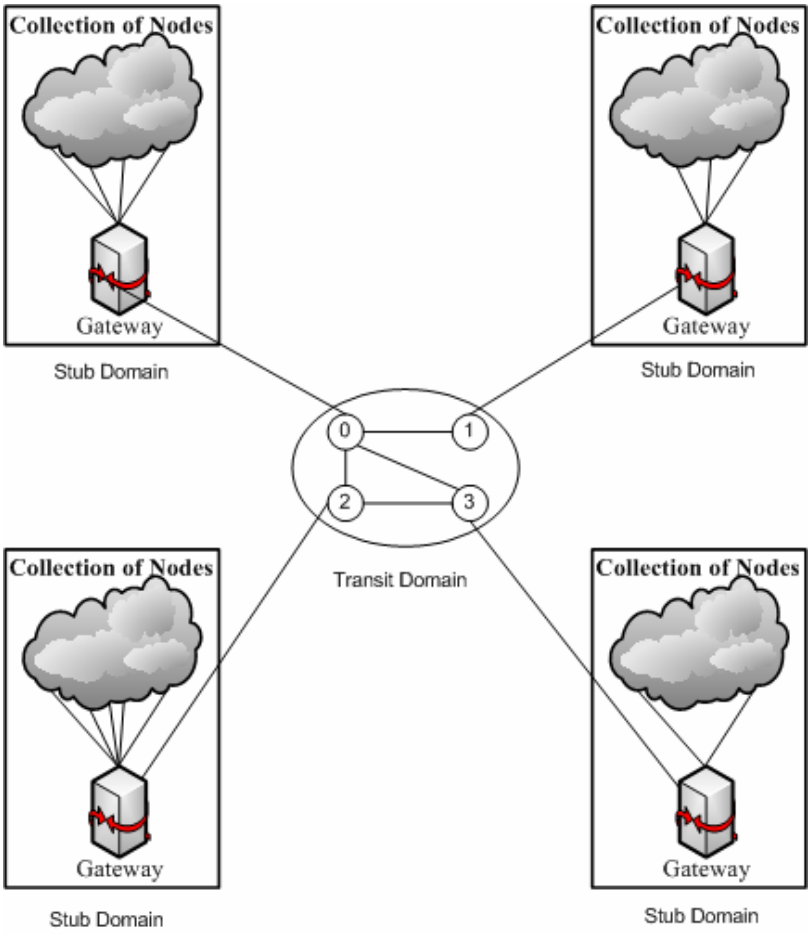


Figure 16. Network Topology.

conversion program from the ns-2. Connections between stub domain nodes were assumed to be duplex links with a bandwidth of 100Mbps. Connections inside the transit domain and the link between transit and stub nodes were assumed to have a bandwidth of 500Mbps. We modified cbrgen.tcl to generate traffic connections inside this topology by making 1000 CBR connections between stub domain nodes. The simulation was then carried out for 2000 seconds, we measured link delays (the amount of time required for a packet to traverse a link considering both bandwidth and propagation delay) between the directly connected nodes in the presence of the random traffic over a 10 second period.

Based on these snapshots, we then determined the communication weights of the network links in the presence of dynamic traffic. Table 1 lists the relevant parameters of the network model used in this study.

Table 1: Network Model Parameters.

Number of Transit Nodes	4
Number of stub nodes/transit node	32
Number of total network nodes	132
Number of total network links	1986
Stub-stub bandwidth	100 Mbps
Transit-transit and transit-stub bandwidth	500 Mbps
Node's processing weight	[20…80] (range)

We ran our tree construction algorithm to create a tree overlay on top of the abovementioned network topology with the application-originating machine as the root node. To create the initial children list, we first went through all network links and made a list for each node $n \in N$, that has direct connections with n. Our tree construction algorithm then finalized the initial children list for each network node n, starting from the root node, ensuring that adding a node to n's children list did not create a cycle. We experimented with different sizes for the children list and found that for the network topology and applications considered in this study, setting the maximum number of children to 5 provided the expected breadth and depth of the tree.

Test Applications

As discussed in Chapter 4, the object graphs extracted by statically analyzing the application code are large and contain nodes and edges that are not necessary for distribution. It becomes time consuming and tedious to manually inspect each graph and select the nodes and the edges that are either application specific or are from the JDK library, but are directly referenced from the application. Despite this, we used two representative applications, one from each target application type and performed static analysis on them to extract the respective object graphs. We then examined the generated graphs and refined them by identifying the useful objects and their relations. The refined graphs were then used as an input to our self-managed deployment algorithm.

Among these applications, the first is the simulation application (App1), which is described in Chapter 1 and is inherently distributed in nature. The application basically instantiates one simulation object, three analysis objects and four visualization objects with communications among them. Figure 17 shows its basic structure and explains its deployment in the underlying network topology. While deploying this application, we used a one-to-one mapping between the application components and the network nodes.

The second application (App2) is basically a master-worker application where 50 independent tasks of equal size need to be performed. This application requires some input data files and produces some output data files. The master node generates the input files, along with the application tasks and also collects the output files. However, to simulate computationally intensive tasks, we kept the size of the input and output files relatively small. While deploying this application we used many-to-one mappings

between the application tasks and the network nodes. Each node in the tree calculates how many tasks it can process per time unit. If there are some tasks left, the node then delegates them to its children. Tasks were delegated to a child according to the capacity of its subtree considering bandwidth, latency, computing speed, etc.

For extensive evaluation of our algorithms and techniques, we also generated some synthetic application graphs with arbitrary structures, vertices and edge weights. More specifically, we generated two graphs that represent inherently distributed applications with 4 and 6 components. For the master-worker type, we generated 9 applications with 10, 20, 30, 40, 60, 70, 80, 90, 100 independent tasks respectively. In the case of the inherently distributed applications, vertices and edges were populated with computational and communication weights in the range [10 … 50] and [5...10] respectively. For the master-worker type, we assumed identical tasks in which the computation to communication ratio is 25:1. For the sake of comparison, we normalized the inferred resource requirements for the vertices and edges of the application graphs generated for App1 and App2 so that they fell within the same range as their synthetic peers.

Utility Function

As explained in the Chapter 6 of this dissertation, we calculated the overall utility of an application graph *G* with priority *p* due to deployment *M*, using the following formulas:

$$U(G,M) = p \times \sum_{(v_j,v_k)\in E} U(v_j, v_k)$$

$$U(v_j, v_k) = \frac{\alpha\, d(n_q)}{f_1(w_g(v_k) \times w_t(n_q)) + f_2(w_g(v_j, v_k) \times w_t(n_p, n_q))}$$

where, $f_1(x) = \beta x$, $f_2(x) = \left(x/\gamma\right)^2$ and $\alpha = 2.5$, $\beta = 0.10$, and $\gamma = 10.0$

Evaluation: Self-Configuration

The purpose of these tests was to evaluate the quality and cost of the initial deployment of the application components to the network nodes, determined by our tree model and autonomic deployment algorithm.

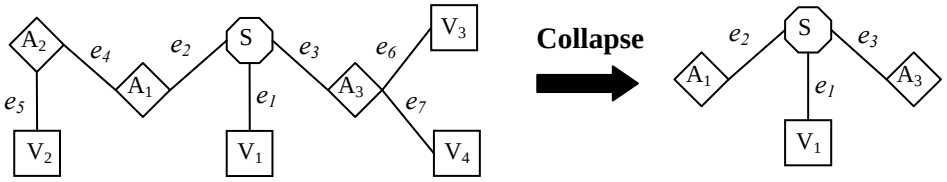
Initial Configuration for the Example Scenarios

The first experiment found the highest utility initial configuration of the graph generated from the App1 on the example tree network (Figure 13). The purpose of this experiment was not only to validate that our approach was able to find the highest utility mapping of an individual application graph to the underlying network, but also to illustrate that the other important aspects of our algorithm — such as graph coarsening, partitioning, etc. — were also working. For this particular experiment, we populated the computational and communication weights of the nodes of the example network (Figure 13) with arbitrary values. Each network node was assigned a computational weight in the range [3.0…15.0] and each link was assigned a communication weight in the range [1…10]. The application graph partitions were sorted according to their sizes and deployed in that order to the highest utility child calculated with the formula specified above. The resultant placements found by the utility function, along with other important

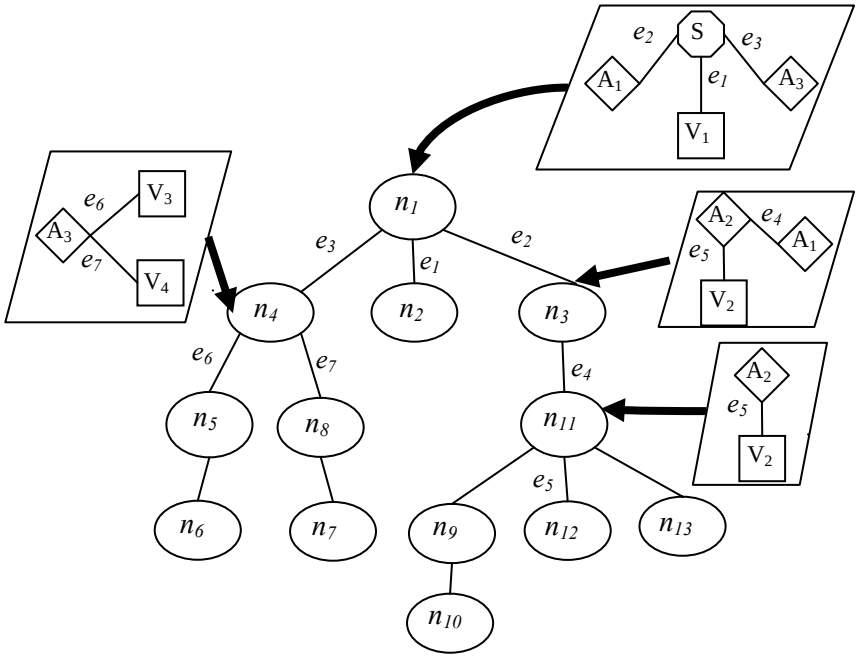
aspects such as graph coarsening, partitioning, propagation etc. are illustrated in Figure 17. This result demonstrates that it is possible to find the most efficient deployment by applying the above utility function locally, between the parent-child pair, and without having the full knowledge of the network and the utility between all pairs of network nodes.

Comparison with the Optimal Scheme

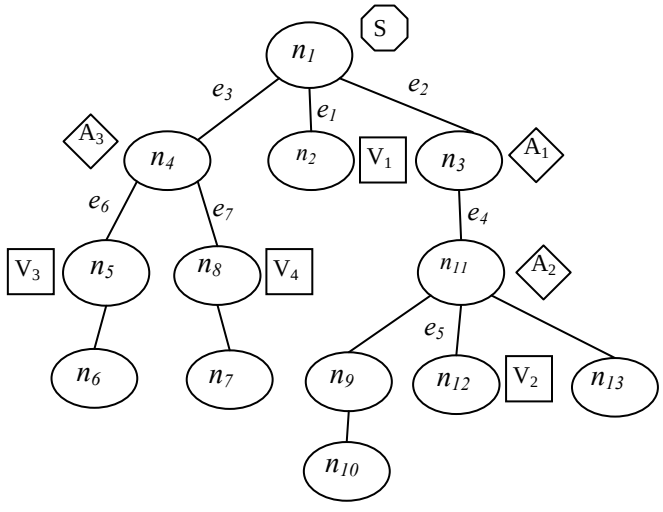
This set of experiments compared the utility and cost of a deployed application graph using optimal schemes based on the original network topology and global knowledge about the system. This is in contrast to our approach that used overlay tree and decentralized deployment decisions based on a minimal amount of locally available knowledge. In the optimal scheme, the assumption is that a central node monitors every computational and communication resource in the system and, based on this global knowledge, makes optimal deployment decisions. However, in centralized approaches the central node becomes a bottleneck with a large number of communications arising from constantly monitoring all the resources in the system. Even if it is possible to gather up-to-date information about all the resources at a central node, finding the optimal deployment means enumerating every possible mapping of the application components to the network resources and selecting the one that produces optimal results. It also grows exponentially with the number of nodes in the network and the number of vertices in the application graph.



(a). Application Graph Collapsed to a Coarser Level.



(b). Graph Partitions are Allocated to the Network Nodes.



(c). Final Placement.

Figure 17. Initial Configuration of the Example Application on the Example Network.

Because of its exponential growth, the abovementioned optimal scheme becomes intractable even for applications with few components, when deployed in the network topology considered for this study. So we developed another semi-optimal scheme that assumed global knowledge but instead of trying every possible mapping, it used a greedy approach to limit the number of cases to evaluate. For both schemes, the network was assumed to be flat, where the central node knew the shortest distance between every pair of nodes in the network and initiated scheduling accordingly. We applied Dijkstra's All Pair Shortest Path algorithm at the central node to calculate the communication weights between every pair of network nodes.

The optimal scheme evaluates the overall utility by testing every possible enumeration of the network nodes, mapping the application components and selecting the mapping M that maximizes the overall utility. The semi-optimal scheme, on the other hand, considers a flat network model and finds the highest utility mapping of individual edges one at a time. For example, to find the highest utility association of edge (v_j,v_k), when vertex v_j is mapped to network node n_p, v_k was mapped exhaustively to all the network nodes available at that time for delegation. The node that resulted in the highest utility was greedily chosen for delegation. The individual maximized utilities for each application edge were then summed together to form the overall utility achieved from the mapping. For the application graph $G = (V,E)$ submitted as input to an arbitrary node $n_p \in N$, where vertex v_j is the origin of the application and is mapped to the node n_p, the algorithm that finds the semi-optimal utility is illustrated as follows:

1. Let overall_utility = 0.0
2. Mark n_p as visited.
3. For each adjacent node v_k of v_j,
    a. For each node $n_q \in N$,
        If n_q is not visited, calculate the utility due to the mapping of the edge (v_j, v_k) to network link (n_p, n_q).
    b. Select the n_q that results in the highest utility in step (a) and configure v_k to that node.
    c. Add the utility achieved due to the mapping of v_k to n_q to the overall_utility.
4. Repeat step 2-3 for each pair of mapping (v_k, n_q) found in the previous step.

The results are presented in Figure 18 and Table 2. Figure 18 compares the utility achieved by all three approaches in case of 4-, 6- and 8-node application graphs and Table 2 reveals the cost associated with them. The network topology (containing 132 nodes) considered for this study was too large to allow an exhaustive search to complete within a reasonable amount of time. So for the optimal scheme, we limited the topology to a single stub domain of 32 nodes. This appears to be a reasonable assumption for the smaller-sized applications considered for this experiment. Yet, we did not have any data beyond an 8-node application in this comparison. After that, determining the optimal mapping became too time-consuming to measure. The results show that the utility achieved by our autonomic approach is on average 30% lower than that of the optimal approach. However, the cost associated with finding the optimal mapping is huge and completely supersedes the benefits of obtaining additional utility by this approach. Figure 18 also illustrates that, in some cases the semi-optimal approach produces lower

utility than our autonomic approach (8-vertex application). Also, in the case of the semi-optimal approach the utility achieved by deploying the 6-vertex application graph is higher than the utility attained in the case of the 8-vertex application graph, which is inconsistent with the fact that utility increases with more deployments.

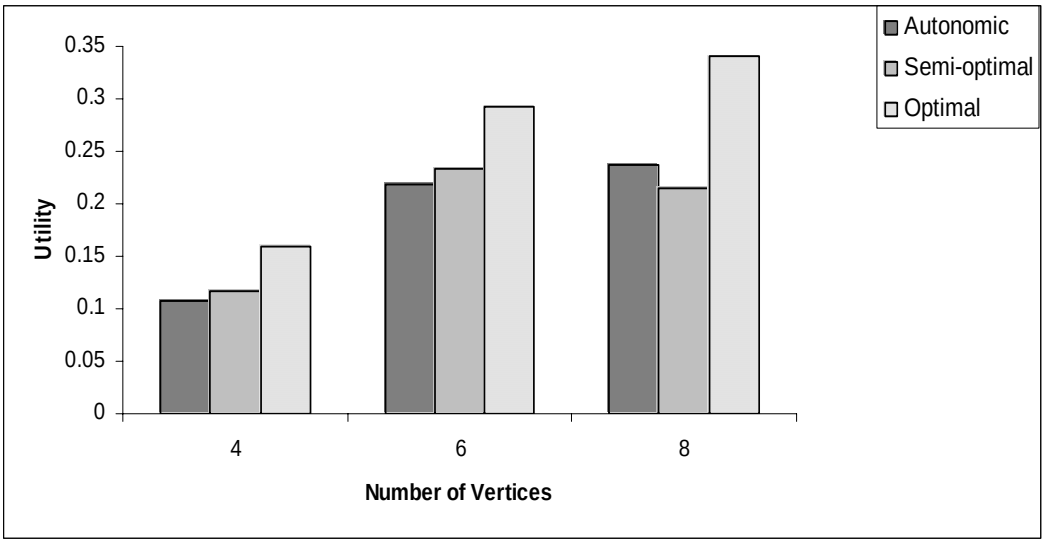


Figure 18. Utilities Achieved by Optimal, Semi-optimal and Autonomic Approaches.

Table 2. Comparison of Deployment Time.

# of Vertices	Autonomic	Semi-optimal	Optimal
4	2 μs	15 μs	8712 μs
6	3 μs	25 μs	15.68 sec
8	3.5 μs	34 μs	1 hour and 39 minute

This observation is explained by the fact that the greedy heuristics applied in the case of semi-optimal deployment does not yield a global optimum. More specifically, it takes a greedy approach to select the best node at each step, considering all the nodes in the network, and there is a possibility that the best utility node found for delegation at a certain stage may already have been delegated in some former stage. However, we will

see later in this section that in general it performs consistently in the case of the master-worker type applications. As a result, we used this approach to compare the results produced by our autonomic algorithm, since we could not use the optimal approach for comparison beyond some smaller programs.

While experimenting with independent task applications, [78] suggested that results may vary depending on tree configurations. In this dissertation, to promote scalability and adaptability, the overlay tree was built without any knowledge of the underlying system. Motivated by the above suggestion, we performed some experiments with different tree configurations, particularly by incorporating some knowledge. For example, while creating the children_list of a certain size for a parent, nodes were included according to their communication costs from the parent. The best utility achieved by experimenting with different tree configurations is only 10% lower than the optimal utility. This suggests that some tree configurations are better than others and that a better deployment can be achieved by exploiting some knowledge about the underlying network. The future work section of Chapter 8 provides some pointers on this issue.

To better realize the tradeoff between utility and cost, we ran the experiment with larger applications, and the results are presented in Figure 19 and

Figure **20**. Figure 19 shows the utility achieved by deploying 10 master-worker type applications in the underlying tree network. Unlike the inherently distributed application the application graph structure does not vary. In a master-worker model, the independent tasks are first allocated to the best network node relative to the root node, then if any remains, to the second best network node, and so on. On the whole, the utility

increases with application size. This is because in a large application more tasks are deployed to more network nodes and therefore the overall utility, which is the sum of all individual utilities, increases.

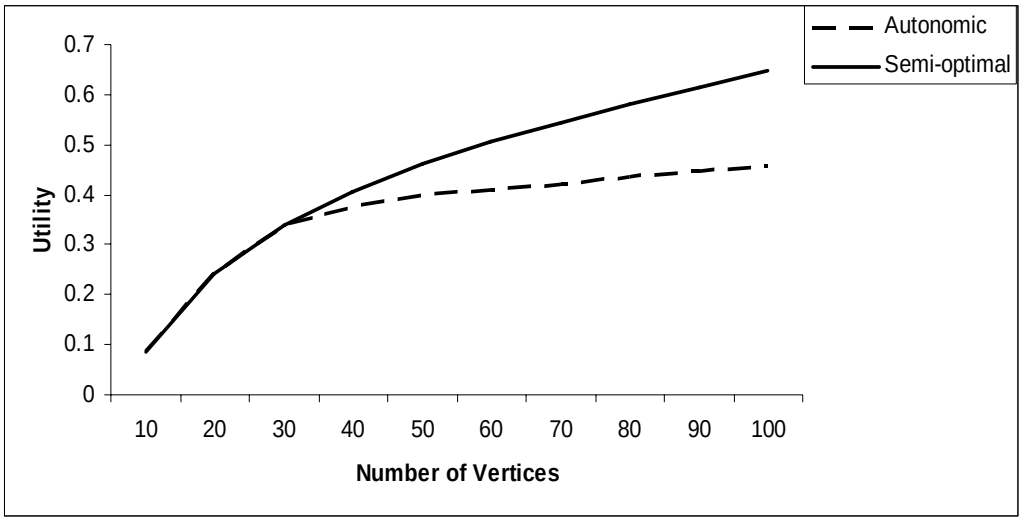


Figure 19. Utilities Attained by Autonomic and Semi-optimal Approaches.

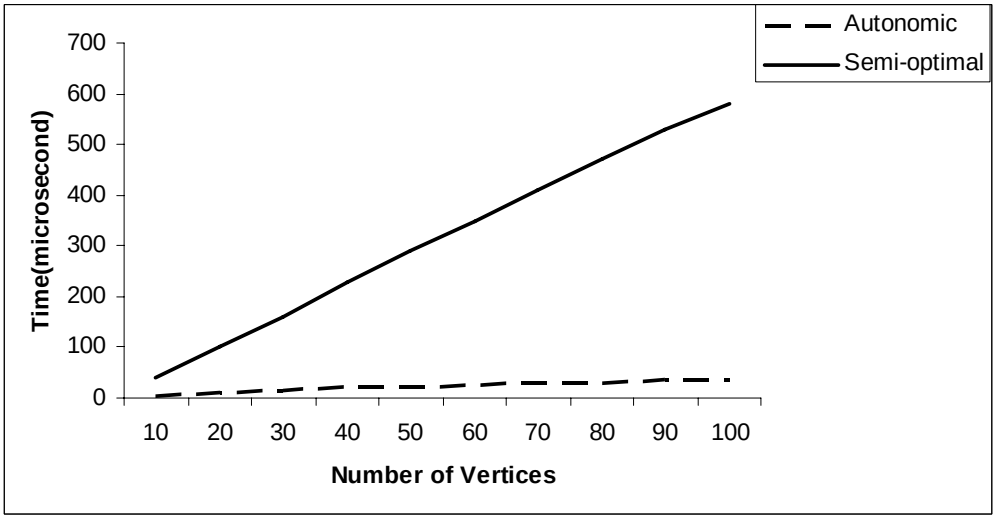


Figure 20. Deployment Time Required for Autonomic and Semi-optimal Approaches.

However in both cases the rate of increase slows with the increase in application size. This happens because the larger application requires more network nodes for

deployment. Gradually they become less efficient, considering the time it takes for the master node to communicate with them. For the autonomic approach, there is another problem. In the tree network, when the independent tasks are propagated down the hierarchy, the associated input files are also propagated with them. If the allocation takes place deep in the tree, all the intermediate nodes must bear the additional communication costs due to the transfer of the input file from the root to the destination node. Therefore, the rate of utility drop is relatively higher for larger applications in the case of an autonomic approach.

Figure 20 shows the time taken by our approach to calculate the initial deployment for an increasing number of application vertices and compares them with the time needed by the semi-optimal approach. On average, compared to the semi-optimal approach, we observe a 92% reduction in the initial deployment calculation time in the autonomic approach. As the application size increases, the cost incurred by our approach is minimal. Hence, it is well suited for larger applications.

Evaluation: Self-Optimization

The purpose of these experiments was to evaluate the effectiveness of the self-optimization in providing an efficient deployment in the presence of various dynamic factors. First, we applied our self-optimizing techniques in the example scenarios to closely monitor the adaptation behavior. We then studied the behavior of our self-managed deployment architecture in the presence of network and processor contention

and when multiple applications were deployed simultaneously. For this set of experiments 8-vertex App1 was used for evaluation.

Dynamic Configuration for the Example Scenarios

This experiment simulated the dynamic reconfiguration of the example scenarios (Figure 17) in the presence of changing network conditions. As illustrated in Figure 17, during initial configuration, the application component V_2 was mapped to the network node n_{12}. Now we focus our attention on node n_{12}, its parent n_{11} and its siblings n_9 and n_{13}. To simulate an increase in network contention in the link ($n_{11,}$ n_{12}) at time t = 30 sec of the execution, we changed the communication cost of this link to a large value. Figure 21 shows that due to this change, the utility drops instantly at the 30th second. However, with self-optimization enabled, this drop in utility was noticed by the monitoring entity running at parent n_{11}, and the system adapted by reconfiguring the application component V_2 to a different network node. As a result, the utility increased again. On the other hand, without self-optimization, the drop in utility was not realized and therefore the initial configuration of component V_2 prevailed and thus the initial utility. During reconfiguration, the parent node n_{11} selected the next best mapping for component V_2. So the utility attained after reconfiguration was not as good as the initial one. The mapping of the application components to the network nodes after reconfiguration is shown in Figure 22.

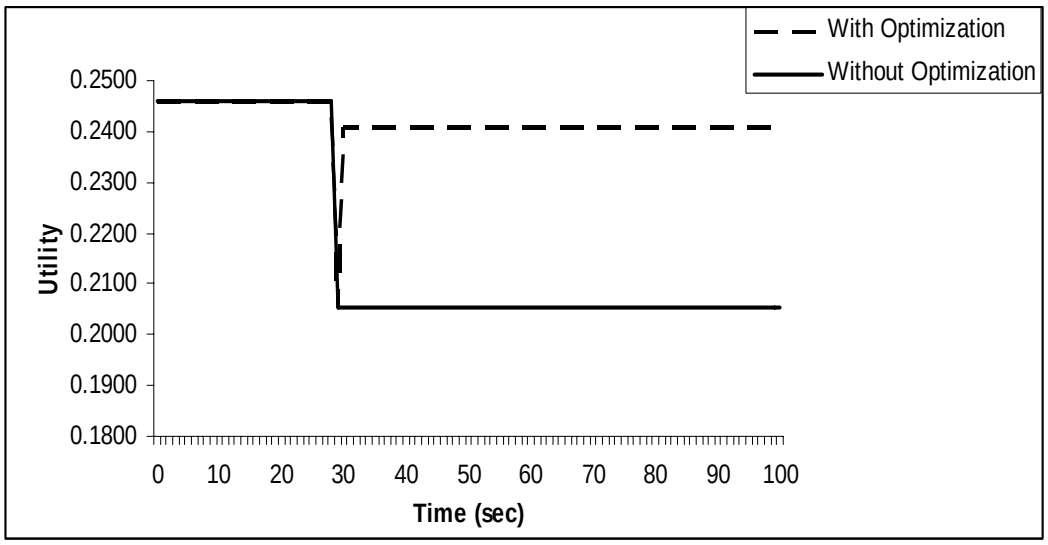


Figure 21. Self-optimization of the Example Scenarios.

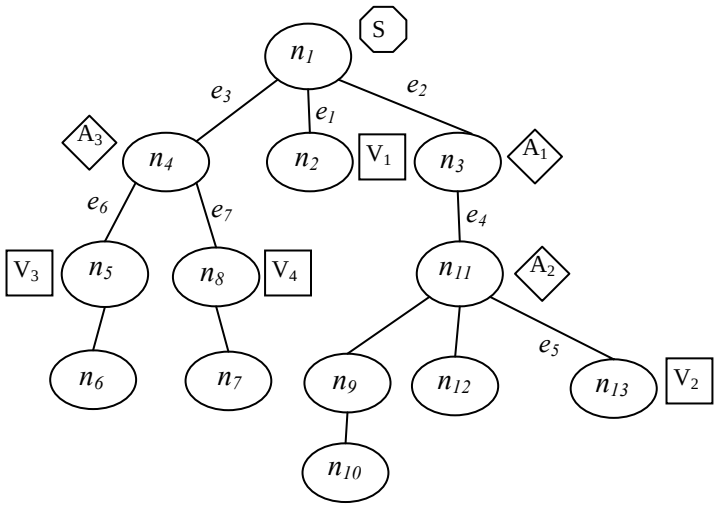


Figure 22. Reconfiguration of the Example Scenarios.

Self-optimization in the case of Network Contention

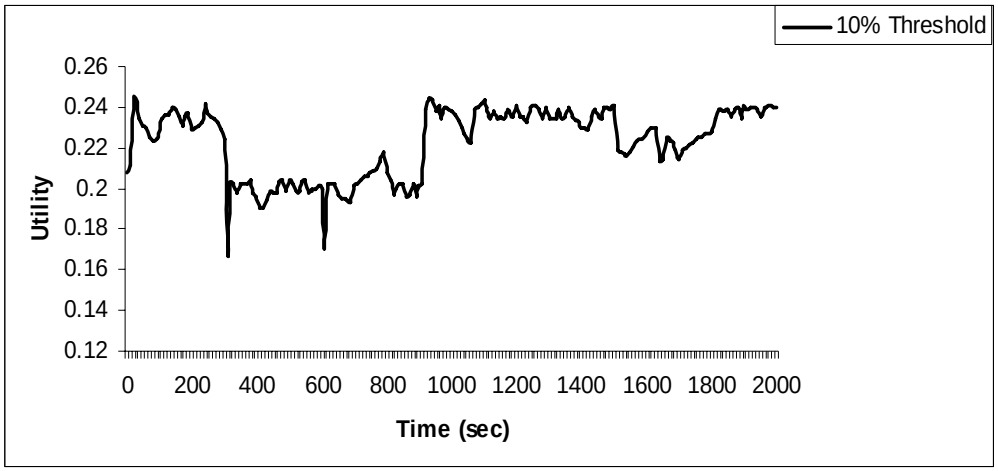
This experiment was conducted to examine the self-optimizing behavior of our architecture in the presence of network contention. The network contention was simulated by introducing cross-traffic among the stub nodes. The optimizing threshold

was determined as the percentage of the initial utility, i.e. when the deviation between the current utility and the best achievable utility was more than, say, 10% of the initial utility, the reconfiguration initiated. By determining threshold in terms of the initial utility instead of specifying some values as a threshold parameter, we allow less knowledgeable users to make better optimization decisions.

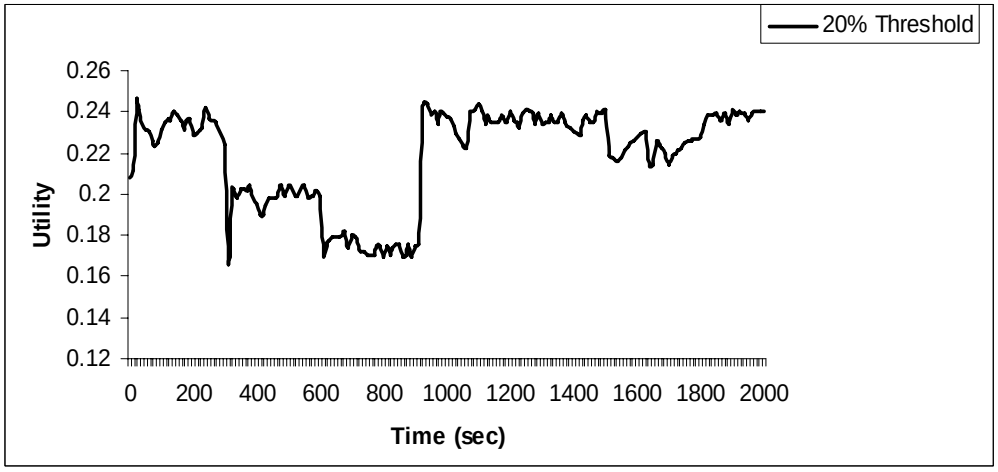
Figure 23 shows the variation of the overall utility with the thresholds set at 10% and 20% of the initial utility both with and without optimization. It is clear from the figure that the utility with reconfiguration is better than without reconfiguration and also lowering the threshold value results in better utility.

Figure 23 (a) demonstrates that with threshold value set at 10%, the optimization routine detected that both communication spikes occurred at time t = 300 seconds and t = 600 seconds and recovered immediately with dynamic reconfiguration. At t = 900 seconds, when a better configuration appeared, the system autonomously switched to the configuration that resulted in the best utility. At t = 1500 seconds, another congestion occurred, however the threshold value was set too high to notice this change in utility and no action was taken.

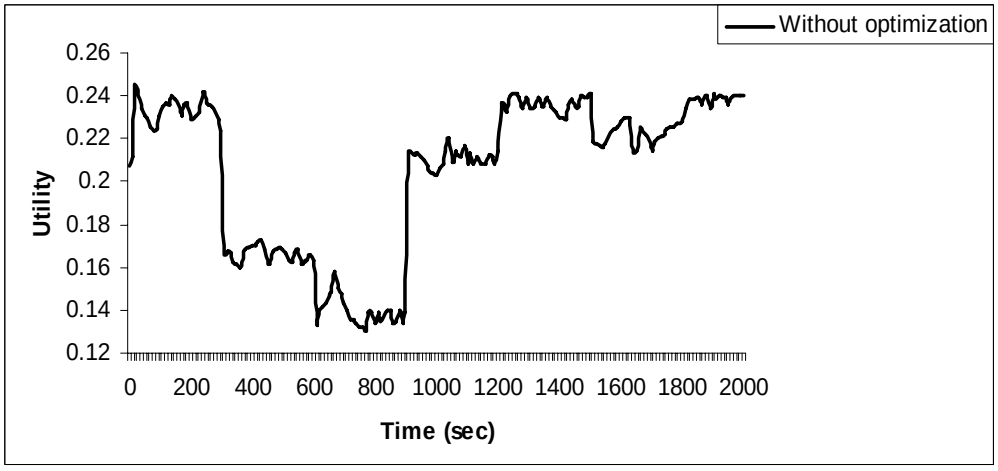
Figure 23 (b), on the other hand, demonstrates the situation for a larger threshold value. At a 20% threshold, the system recovered from the congestion at t = 300 seconds, but not from the one that occurred at t = 600 seconds. Therefore, the reconfiguration, which was performed after t = 300 seconds, continued until t = 900 seconds, when a better configuration appeared. Figure 23 (c) shows the change in utility when no optimization is applied, and as a result the initial configuration continued until the end of



a) Optimization with 10% Threshold.



b) Optimization with 20% threshold.



c) Without Optimization.

Figure 23. Utility Variation in the Presence of Network Contention.

the simulation. On average, the 10% threshold results in 15% more utility and the 20% threshold results in 11% more utility when compared to the utility achieved without any optimization.

Self-optimization in the case of Processor Overload

This experiment was performed to study the self-optimizing behavior of our system in the presence of the overloaded processor. We could not find any way to simulate processor contention in the ns-2 simulator, so we changed the processing weights of some of the nodes in the middle of the simulation. The utility variation due to the increase in processor load is depicted in Figure 24. Again with the self-optimization, a better utility is achieved.

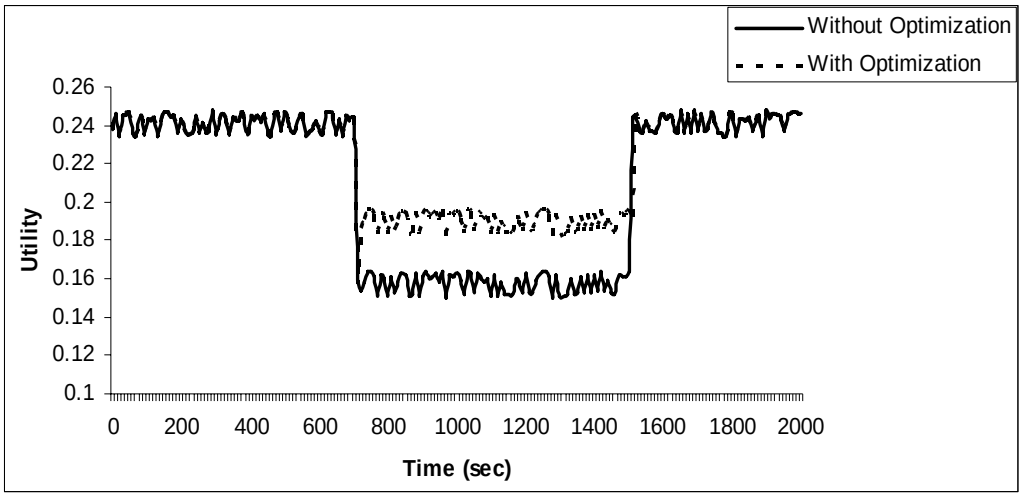


Figure 24. Utility Variation in the Presence of Overloaded Processors.

Self-optimization in the presence of Multiple Applications

This experiment was designed to examine the self-optimizing behavior of our system in the presence of multiple applications with different priorities. For this experiment, except for 8-vertex App1, we used another similar application with eight

components. The priority of App1 was set to 1 and was introduced to the system at t = 0 seconds. At t = 450 seconds, the second application with priority 3 was inserted into the system. Figure 25 shows the changing utility due to the deployment of multiple applications with different priorities. The utility of the lower priority application graph decreases significantly once the higher priority graph is injected into the system. This is because preference was given to the higher priority graph when allocating system resources, as in that way, the overall utility was maximized. Thus, the most efficient network nodes and links were assigned to the higher priority graph, where they had previously been utilized by the lower priority one. Hence, the utility for the lower priority graph was reduced.

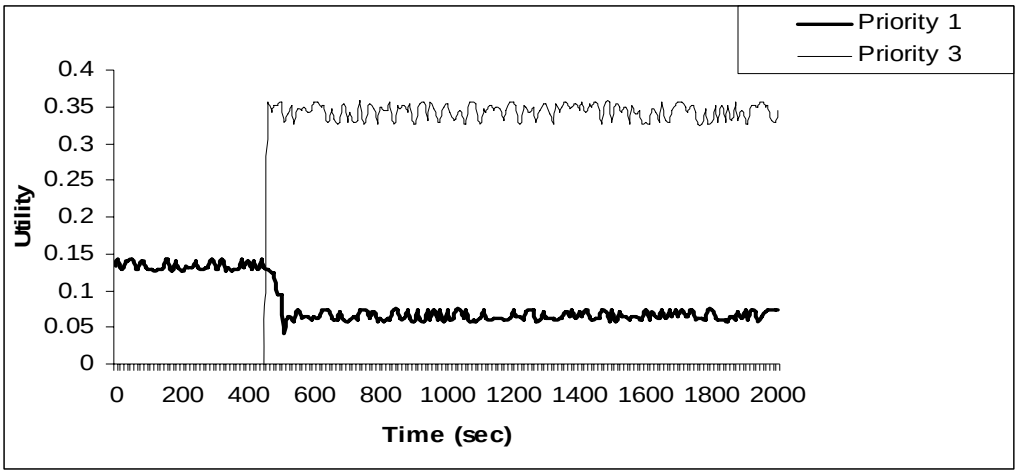


Figure 25. Utility Variation in the Presence of Multiple Applications.

Summary of the Results

In general, the above experiments show the potential of our self-managed deployment architecture for effectively placing the application components in the

network nodes and for coping with underlying changes in the computing platform. Our first experiment with the small size example network allows us to closely monitor the initial deployment process, and the resultant placements validate that our autonomic deployment algorithm can autonomously find the highest utility association among the set of application components and the set of distributed nodes autonomously.

The next experiment compares the utility achieved by the autonomic approach to the optimal utility, which is achieved by allowing a central node to try all possibilities exhaustively before selecting the best mapping. The results show a tradeoff between the optimal utility and the time to achieve it. On average, with the autonomic approach, the percentage loss of utility is less than 30% compared to the optimal utility, with an enormous drop in the time needed to compute the deployment. The results also reveal that, even with the huge deployment time, we can only deploy a few small-sized applications using exhaustive search.

To evaluate the robustness of our algorithm for large applications, we also performed experiments that deployed applications with increasing numbers of components and compared the resultant utility and time to achieve the deployment with the semi-optimal approach. The results show that in both cases the rate of increase of the utility slows with the increase in the application size. This is expected, since with a large number of application components, some have to be mapped to the nodes that are far away from the root node. So for them, the communication cost increases, eventually leading to lower overall utility. However, as the application size grows, we observe a

92% reduction in the deployment time of our autonomic approach compared to the semi-optimal approach.

The next experiments evaluate the self-optimizing behavior that is driven by the change in utility value. The results show that there is almost instantaneous change in utility value when an attribute changes. Also based on the reconfiguration threshold, the system autonomously reassigns the network nodes to achieve a better utility. The results suggest that an appropriate threshold value can be used to tradeoff utility for a lower number of reconfigurations. The self-optimization is applied in the presence of an overloaded processor, network contention and multiple application graphs. In all cases, our autonomic approach results in better utility with optimization than without optimization. The most important implication of this result is that, by properly formulating and parameterizing a utility function, it is possible to trigger and achieve a desired level of self-optimization for a wide variety of applications, systems and user specific attributes.

CHAPTER 8

CONCLUSION AND FUTURE WORK

Contributions

In this dissertation we investigate self-managed deployment of computationally intensive scientific and engineering applications in a highly dynamic distributed environment consisting of non-dedicated heterogeneous computers. Our goal is to achieve self-adaptive, cost-effective and scalable deployment that results in higher utilization of the distributed resources while meeting application specific performance demands. To support our goal, in this dissertation we made the following key contributions.

- To transform an application to a self-managed distributed one, we realized the importance of having a program's run time structure abstracted into a graph model that could act as a basis for self management and self optimization decisions. We then presented the design and implementation of an algorithm that statically analyzes the application code to identify the runtime instances and their interactions. To infer the computational and communication resource requirements of the application components and their links, we adopted several techniques based on measurements and heuristics. The resource requirements are estimated at the same time the application graph is being constructed. Thus, this approach does not incur additional cost due to the repetitive analysis of the application code.

- We modeled the target distributed environment as a tree, where the execution starts at the root node of the tree and computation then organizes itself on the available nodes according to a pattern that emerges from their point-to-point interactions. The most important aspect of this design choice is that it promotes scalability and adaptability by decentralizing the resource monitoring and the utility calculation. The overlay tree network was built on top of the existing topology and techniques were developed to handle related issues, such as the initial children list, presence of the broken links or cycles in the tree, etc.
- We designed and implemented an autonomic algorithm that autonomously deploys the application graph in the tree overlay in a utility-aware way. The deployment was governed by a utility function, which was designed to express the system's overall utility based on a wide variety of application, system and user-specific attributes. By exploiting the application graph, tree overlay network and the utility function, the autonomic algorithm found the best scheduling based solely on the minimal knowledge locally available at each node. Our experiments demonstrated that it is possible to find an efficient deployment by applying the utility function locally, between the parent-child pair, and without having the full knowledge of the network and the utility between all pairs of network nodes. We also showed that by decentralizing the problem of finding the highest utility association among the set of components and the set of nodes, our solution makes self-management feasible in the case of an infrastructure containing a large number of nodes and in larger applications.

- We also incorporated techniques into our scheduling strategy to tackle the platform dynamics, i.e. where resources exhibit dynamic performance characteristics and availability. In our design, each parent node in the tree monitors the computation and communication speed of its children. Utility is calculated recurrently based on this monitored information, and optimization is driven by the changed utility. We assessed the robustness of the self-managed deployment architecture under variable workloads, network contentions and in the case of the simultaneous deployment of multiple applications. Our experimental results demonstrated that, in all cases, the utility achieved by our autonomic approach is better than the utility achieved without optimization, and resources are reconfigured autonomously to attain the best utility.

Several challenges need to be dealt with long before the vision of autonomic computing becomes a reality. These challenges include a wide range of scientific, engineering, technical, and architectural aspects of self-manageable systems. This dissertation addresses some of the challenges of autonomic systems. The self-managed deployment architecture presented in this dissertation is scalable and adaptable and thus ideally suited for the deployment and management of large applications in currently emerging computing platforms. We believe that the techniques and algorithms presented in this dissertation are valuable in closing the gap between the vision of a true autonomic system and its practical implementation.

Future Work

In this dissertation we have developed techniques and algorithms to achieve self-managed deployment and evaluated them in the case of a simulated environment. In the future we would like to implement prototype software, which uses object graphs, tree models, autonomic scheduling and dynamic reconfiguration, to deploy a wide variety of applications on real computing platforms.

As discussed in Chapter 4, the size of the object graphs generated by our static analysis approach is significant, and they contain objects and relations that are not necessary for our purpose. Therefore, the object graphs need some further refinement to reduce them to a manageable size while preserving useful information needed for the further analysis. A possible solution for this problem is to incorporate dynamic information, based on offline profiling, into the static analysis to enhance the object graph. The object graph size can be reduced significantly if we trim off the objects that are not actually executed in the profiled run. However, a profiled run may differ from the actual run, depending on various dynamic factors but the difference is expected to be relatively small. A profiled run will also help to estimate the node and edge weights more accurately by incorporating runtime information, such as loop counts and conditional branch directions. We would like to explore this idea in the future.

We are also planning to investigate the effect of the tree overlay network on the performance in detail. Our experiments show that utility varies depending on the configuration of the tree network. Reference [91] suggested a mechanism where the tree is continuously restructured during the execution of the application, such that high

performance and lower bandwidth nodes are always near the root. The adaptation in their model is achieved by the nodes using passive feedback to monitor the performance of their children. We would like to pursue this and the other potential approaches to restructure the tree so that it best utilizes the system.

In this dissertation, we considered dynamism in the computing environments. In the future we would like to extend our approaches to handle dynamic changes in the application structure and requirements. In the context of Service Oriented Architectures (SOAs), as new services are added to or removed from the application, new components are started or stopped. For some applications, the workload may vary widely during execution resulting in dynamically changing resource requirements for these applications. In order to handle this variability in application structure and resource requirements, we need to widen our self-managed deployment approach to adapt to the changes in application.

In addition to the static analysis-based resource requirement inference, we also would like to be able to infer application needs from their quality of service (QoS) specification. Typically, the Service Level agreements (SLAs) allow the users to specify their desired QoS, which must be maintained by the service providers. Examples of QoS specifications include bounds on average response time and throughput. In the future, we plan to extend our architecture so that the deployment can be done to meet the application's QoS requirements.

We are also interested in quantitative evaluation of self-managed systems along other dimensions besides performance. To cope with the ever-increasing complexity

involved in today's systems, the focus has now shifted from their performance to other system's properties, such as robustness, availability, manageability, usability. These properties are becoming as important as optimal performance. However, quantifying and modeling these properties is challenging and has not yet been addressed. We are interested in applying a combination of experimental and analytic techniques to develop metrics and benchmarks that can be used to evaluate these properties. Quantifying those properties would be a significant step in developing truly self-managing systems.

REFERENCES

1. D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. In *Proceedings of the Second International Conference on Peer-to-Peer Computing*, pages 1–51, 2002.

2. H. S. Nwana. Software Agents: An Overview. *Knowledge Engineering Review*, Vol. 11, No. 3, pages 1–40, 1996.

3. M. P. Singh (Ed). Practical Handbook of Internet Computing. Chapman & Hall/ CRC Press, 2004.

4. I. Foster, C. Kesselman (Ed). The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, 2003.

5. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. In *IEEE Computer*, Vol. 36, No. 1, pages 41–50, 2003.

6. A. Ganek and T. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal,* Vol. 42, No. 1 pages 5-18, 2003.

7. IBM Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

8. S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar and H. Liu. The Autonomic Computing Paradigm, *Cluster Computing*, Vol. 9 No.1, pages 5-17, 2006.

9. R. Murch. Autonomic Computing. *Prentice-Hall*, 2004.

10. M. J. Oudshoorn, M. M. Fuad and D. Deb. Towards an Automatic Distribution System - Issues and Challenges. In *Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 399-404, 2005.

11. M. M. Fuad and M. J. Oudshoorn. An Autonomic Architecture for Legacy Systems. In *Proceedings of the Third IEEE Workshop on Engineering of Autonomic Systems (EASe)*, pages 79-88, 2006.

12. M. M. Fuad. An Autonomic Software Architecture for Distributed Applications. *Ph.D. thesis*, Department of Computer Science, Montana State University, USA, 2007.

13. M. M. Fuad, D. Deb and M. J. Oudshoorn. An Autonomic Element Design for a Distributed Object System. In *Proceedings of the ISCA 20th International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 273-279, 2007.

14. W. E. Walsh, G. Tesauro, J. O. Kephart, R. Das. Utility Functions in Autonomic Systems. In *Proceedings. Of the First International Conference on Autonomic Computing (ICAC),* pages 70-77, 2004.

15. SETI@home. http://setiathome.ssl.berkeley.edu, 2001.

16. D. F. Sittig, D. Foulser, N. Carriero, G. McCorkle, P. L. Miller. A Parallel Computing Approach to Genetic Sequence Comparison: The Master-worker Paradigm with Interworker Communication. *Computers and Biomedical Research*, Vol.24 No.2, pages 152-169, 1991.

17. J. Cowie, B. Dodson, R. Elkenbrach-Huizing, A. Lenstra, P. Montgomery and J. Zayer. A World Wide Number Field Sieve Factoring Record: On to 512 Bits. *Advances in Cryptology*, Vol. 1163 of LNCS, pages 382–394, 1996.

18. M. Waterman and T. Smith. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, Vol. 147, pages195–197, 1981.

19. P. S. Churchland and T. J. Sejnowski. The Computational Brain, MIT Press, 1992.

20. Hewlett-Packard. Infrastructure and Management Solutions for the Adaptive Enterprise.
http://www.hp.com/products1/promos/adaptive_enterprise/pdfs/vision_for_ae.pdf

21. Microsoft. Microsoft Dynamic Systems Initiative Overview.
http://www.microsoft.com/windowsserversystem/dsi/dsiwp.mspx

22. Intel. Autonomic Platform Research (APR).
http://www.intel.com/cd/ids/developer/asmona/eng/192589.htm

23. Oracle. Oracle Automatic Workload Repository (AWR).
http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php

24. J. O. Kephart, R. Das. Achieving Self-Management via Utility Functions. *IEEE Internet Computing*, Vol. 11, No. 1, pages 40-48, 2007.

25. H. Liu and M. Parashar. Accord: A Programming Framework for Autonomic Applications. *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems*, Editors: R. Sterritt and T. Bapty, IEEE Press, Vol. 36, No. 3, page. 341 – 352, 2006.

26. M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang and S. Hariri. AutoMate: Enabling Autonomic Grid Applications. *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, Special Issue on Autonomic Computing, Kluwer Academic Publishers, Vol. 9, No. 1, 2006.

27. X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. AUTONOMIA: An Autonomic Computing Environment. In *Proceedings of the 2003 IEEE International Performance, Computing, and Communication Conference*, pages 61-68, 2003.

28. D. M. Chess, A. Segal, I. Whalley and S. R. White. Unity: Experiences with a Prototype Autonomic Computing System. In *Proceedings of the First International. Conference on Autonomic Computing (ICAC)*, pages 140-147, 2004.

29. J. Parekh, G. Kaiser, P. Gross and G. Valetto. Retrofitting Autonomic Capabilities onto Legacy Systems. *Journal of Cluster Computing*, Kluwer Academic Publishers, Vol. 9, No. 2 pages 141-159, 2006.

30. S. Cheng, A. Huang, D. Garlan, B. Schmerl, P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In *Proceedings of the First International Conference on Autonomic Computing (ICAC)* pages 276-277, 2004.

31. R. V. Renesse, K. P. Birman, W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems (TOCS)*, Vol. 21 No. 2, pages164-206, 2003.

32. K. Schwan et al. Autoflow: Autonomic Information Flows for Critical Information Systems. Autonomic Computing: Concepts, Infrastructure, and Applications, CRC Press, 2006.

33. V. Kumar, B. F. Cooper, K. Schwan. Distributed Stream Management using Utility-Driven Self-Adaptive Middleware. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*, pages 3-14, 2005.

34. M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, Vol. 9, No. 11, pages 1125-1242, 1997.

35. M. Dahm. Doorastha—a step towards distribution transparency. *JIT,* 2000.

36. A. Spiegel. Automatic Distribution of Object-Oriented Programs. *PhD thesis*, Fachbereich Mathematik u. Informatik, Freie Universitat, Berlin, 2002.

37. M. Tatsubori, T. Sasaki, S. Chiba and K. Itano. A Byte-code Translator for Distributed Execution of Legacy Java Software. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 236-255, 2001.

38. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* 2002.

39. G. C. Hunt, and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI)*, pages 187-200, 1999.

40. R. E. Diaconescu, L. Wang, Z. Mouri and M. Chu. A Compiler and Runtime Infrastructure for Automatic Program Distribution. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

41. Y. Aridor, M. Factor, and A. Teperman. CJVM: a Single System Image of a JVM on a Cluster. *ICPP*, 1999.

42. H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared-Object System. *ACM Transaction on Computer Systems*, Vol. 16, No.1, pages 1-40, 1999

43. J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152-164, 1991.

44. W. Yu, and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, Vol. 9, No. 11, pages 1213-1224, 1997.

45. S. M. Sadjadi and P. K. McKinley. A Survey of Adaptive Middleware. *Technical Report MSU-CSE-03-35*, Computer Science and Engineering, Michigan State University, 2003.

46. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 1241, 1997.

47. G. Kiczales et al. An Overview of AspectJ. In *Proceedings of European Conference on Object-Object Programming (ECOOP)*, pages 327–353, 2001.

48. H. Kim. AspectC#: An AOSD implementation for C#, *Masters Thesis*, Department of Computer Science, Trinity College, Dublin, 2002.

49. D. Lafferty et al. Language Independent Aspect-Oriented Programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 1–12, 2003.

50. B. Rasmussen et al. Aspect.NET - A Cross-Language Aspect Weaver. Department of Computer Science, Trinity College, Dublin, 2002.

51. A. Frei et al., A Dynamic AOP-Engine for .NET, *Technical Report 445*, Department of Computer Science, ETH Zurich, May 2004.

52. F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckmeier. AspectIX: An Aspect-Oriented and CORBA-Compliant ORB Architecture. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 1998.

53. J. Lam. CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime Demonstration at the *First International Conference on Aspect-Oriented Software Development*, 2002.

54. M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A Class-based Macro System for Java. In *Proceedings of OORaSE*, pages 117–133, 1999.

55. E. P. Kasten and P. K. McKinley. Adaptive Java: Refractive and Transmutative Support for Adaptive Software. *Technical Report MSU-CSE-01-30*, Computer Science and Engineering, Michigan State University, 2001

56. V. Adve, V. V. Lam, and B. Ensink. Language and compiler support for adaptive distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001),* 2001.

57. D. C. Schmidt, D. L. Levine, and S. Mungee. The Design of the TAO Real-time Object Request Broker. *Computer Communications*, Vol. 21, pages 294–324, 1998.

58. F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalh˜aes, and R. H. Campbell. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, 2000.

59. G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing ((Middleware'98)*, 1998.

60. J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, Vol. 3, No. 1, 1997.

61. R. Koster, A. P. Black, J. Huang, J.Walpole, and C. Pu. Thread Transparency in Information Flow Middleware. In *Proceedings of the International Conference on Distributed Systems Platforms and Open Distributed Processing*, Springer Verlag, 2001.

62. R. Baldoni, C. Marchetti, and A. Termini. Active Software Replication Through a Three-tier Approach. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, pages 109–118, 2002.

63. Sun Microsystems, http://java.sun.com/products/ejb/, Enterprise JavaBeans Technology, 2001.

64. Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/99-10-05, CORBA Components Model - FTF drafts for MOF chapter.

65. B. Redmond and V. Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behavior. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.

66. M. Golm and J. Kleinoder. metaXa and the Future of Reflection. In *Proceedings of Workshop on Reflective Programming in C++ and Java*, pages 1–5, 1998.

67. A. Oliva and L. E. Buzato. The Implementation of Guaran´a on Java. *Technical Report IC-98-32*, Universidade Estadual de Campinas, Sept. 1998.

68. A. Popovici, T. Gross, and G. Alonso. Dynamic Homogenous AOP with PROSE. *Technical Report*, Department of Computer Science, Federal Institute of Technology, Zurich, 2001.

69. M. J. Oudshoorn, M. M. Fuad and D. Deb. Towards Autonomic Computing: Injecting Self-Organizing and Self-Healing Properties into Java Programs. New Trends in Software Methodologies, Tools and Techniques, Volume 147 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pages 384-406, 2006.

70. D. Deb, M. M. Fuad and M. J. Oudshoorn. Towards Autonomic Distribution of Existing Object Oriented Programs. *International Conference on Autonomic and Autonomous Systems (ICAS)*, IEEE Press, pages 17-23, 2006.

71. Sable research group, www.sable.mcgill.ca/soot.

72. O. Lhotak. Spark: A Flexible Points-to analysis Framework for Java. *Master's thesis,* McGill University, December 2002.

73. A. Camesi, J. Hulaas, and W. Binder. Continuous Bytecode Instruction Counting for CPU Consumption Estimation. In *Proceedings of the Third International Conference on the Quantitative Evaluation of Systems (QEST '06)*, pages 19–30, 2006.

74. H. Bal, F. Kaashoek, and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transaction on Software Engineering.* Vol. 18, No. 3, pages190-205, 1992.

75. F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, Vol. 14, No.4, pages 369--382, 2003.

76. A. S. Grimshaw, W. A. Wulf, and the Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, Vol. 40, No. 1, pages 39-45, 1997.

77. O. Beaumont, A. Legrand, Y. Robert, L. Carter, J. Ferrante. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

78. B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous Protocols for Bandwidth-centric Scheduling of Independent-task Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

79. C. Banino. A Distributed Procedure for Bandwidth-Centric Scheduling of Independent-Task Applications. In *Proceedings of the* IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005.

80. M. Ripeanu, I. Foster and A. Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design, *IEEE Internet Computing*, Vol. 6, No. 1, February 2002.

81. S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. A Scalable Content-Addressable Network. *In Proceedings of ACM SIGCOMM 2001*.

82. D. Deb, M. M. Fuad and M. J. Oudshoorn. ADE: Utility Driven Self-management in a Networked Environment. *Journal of Computers (JCP)*, Academy Publishers, Vol. 2, No. 9, 2007.

83. D. Deb and M. J. Oudshoorn. On Utility Driven Deployment in a Distributed Environment. In *Proceedings of* Fourth *IEEE Workshop on Engineering of Autonomic Systems (EASe 2007)*, pages 14-23, 2007.

84. G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs, *Journal of Parallel and Distributed Computing*, Vol. 48, pages 86-129, 1998.

85. D. Deb, M. J. Oudshoorn and J. Paxton. Self-Managed Deployment in a Distributed Environment via Utility Functions. *20th International Conference on Software Engineering and Knowledge Engineering (SEKE08)*, submitted for publication.

86. Intel Topology Generator. http://topology.eecs.umich.edu/inet.

87. BRITE: Boston University Representative Internet Topology gEnerator. http://www.cs.bu.edu/brite.

88. Tiers Topology Generator. http://www.isi.edu/nsnam/ns/ns-topogen.html

89. GT-ITM: Georgia Tech Internetwork Topology Models http://www.cc.gatech.edu/projects/gtitm.

90. The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns.

91. A. J. Chakravarti, G. Baumgartner and M. Lauria. The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network. *In Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 96-103, 2004.