Achieving Self-managed Deployment in a Distributed Environment via Utility Functions

By

Debzani Deb

A dissertation proposal submitted as part

of the requirements for the degree

of

Doctor of Philosophy

In Computer Science

MONTANA STATE UNIVERSITY

Bozeman, Montana

January, 2008

ABSTRACT

By harnessing the computational power of distributed heterogeneous resources, it is possible to build a large scale integrated system so that a centralized program is partitioned and distributed across those resources in a way that maximizes the system's overall utility. However, building such a system is a staggering challenge because of the associated complexities. This paper proposes a self-managing distributed system ADE (Autonomic Distributed Environment), which engages autonomic elements to automatically take an existing centralized application and distribute it across available resources. The autonomic elements provide self-management to handle the complexities associated with distribution, configuration, coordination and efficient execution of program components. The proposed approach models a centralized application in terms of an application graph consisting of application components and then deploys the application components across the underlying utility-aware hierarchically organized distributed resources so that all constraints and requirements are satisfied and the system's overall utility is maximized. Then, based on the observations obtained by the monitoring of the system resources, ADE redeploys the application graph to maintain maximized system utilization in spite of the dynamism and uncertainty involved in the system.

TABLE OF CONTENTS

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 independent 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 require massive computational power and resources and can benefit from such an integrated infrastructure.

In most corporations, research institutes or universities, there are significant numbers of computing resources 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 formed. With such an infrastructure, it is possible to solve computationally intensive problems efficiently in a cost effective manner as opposed to replacing these systems with expensive computational resources such as supercomputers. Having such a large-scale system, one can effectively partition an existing centralized application in terms of communicating components and distribute those components among the available resources in a manner which results in the efficient execution of user program and maximizes resource utilization. This was 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, there are many challenging aspects associated with effectively partitioning large-scale applications into several components as well as the mapping and scheduling of those components over the heterogeneous resources across the system. Firstly, the programmer wishing to execute such an application may not have necessary skills to rewrite the application to achieve effective partitioning and distribution across the network. To transform a regular, centralized application into a distributed one, the programmer needs to perform a large number of changes and most of these changes require thorough knowledge of both the application structure and the underlying

architecture where the application is going to be deployed. Secondly, ensuring maximum utilization requires mechanisms to estimate the application component's computational and communication needs and their interdependencies so that an efficient mapping of components to resources can be achieved and the mapping's overall communication cost is minimized. Thirdly, application behavior is highly dynamic and a different distribution configuration may be appropriate in different phases of the execution of an application. As a consequence, application components should be easy to migrate at runtime to enhance locality and to minimize communication cost. Finally, the application components should be tailored to dynamically respond to their environment by expanding their functionality or enhancing their performance as the underlying infrastructure is heterogeneous and changes over time. Each of these factors introduces additional complexities. In order to deal with them the system must be adaptive and dynamic in nature.

It would be tremendously useful to have a system that can automatically transform an existing application into a distributed one without the programmer being concerned about distribution and management issues, and which can deploy the distributed application across a large-scale integrated infrastructure efficiently. The complexity and cost associated with the management of such an infrastructure and the necessary enhancement of an existing application such that they deploy effectively on that dynamic infrastructure is significant. Therefore, automation is paramount in order to lower operation costs, to allow developers to largely ignore complex distribution issues, to manage system complexities and to maximize overall utilization of the system. This research envisions such an automatic system as an autonomic computing challenge [5,6].

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*: An autonomic computing system must be able to dynamically configure and reconfigure itself based on its state and the state of its execution environment.
- *Self-optimizing*: An autonomic system should have the capability of maximizing resource allocation and utilization for satisfying user requests and application demands.
- *Self-healing*: An autonomic system must detect a failed component, eliminate it, or replace it with another component without disrupting the system. On the other hand, it must predict problems and prevent failures.
- *Self-protecting*: An autonomic system should be capable of detecting and protecting its resources from both internal and external attack and maintaining overall system security and integrity.

This proposal presents ADE, Autonomic Distributed Environment, a self-managed distributed system which engages autonomic elements to automatically take an existing centralized application and distribute it across available resources. The autonomic elements provide self-management capabilities to handle the complexities associated with distribution, configuration, coordination and efficient execution of program components. The proposed approach models a centralized application in terms of an application graph consisting of application components and then deploys the application components across the underlying utility-aware hierarchically organized distributed resources so that all constraints and requirements are satisfied and the system's overall utility [8] is maximized. Then, based on the observations obtained by the monitoring of the system resources, ADE redeploys the application graph to maintain maximized system utilization in spite of the dynamism and uncertainty involved in the system.

The remainder of this proposal is organized as follows. The rest of this chapter discusses the challenges needed to be met in order to design a self-managing distributed system, the goals of this research, the scope and the assumptions that have been made for

this research. Chapter two provides background information on some important concepts related to this research and also discusses the relevant research initiatives undertaken at different universities and industries. Chapter three gives an overview of the research plan detailing different design, implementation and evaluation aspects.

1.1 Challenges in Designing Self-Managing Distributed System

High performance applications are playing an increasingly important role in the scientific and engineering research community. Typically these applications are inherently multi-phased, highly dynamic, composed of a large number of software components along with dynamic interactions among the components. The proposed distributed infrastructure 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. Some of the challenges ADE must meet are detailed as follows:

- **Distributed Application Development:** For an average user, the task of distributing a large computation across a dynamic, unpredictable and heterogeneous environment proves to be a tedious and cumbersome job. Programmers wishing to do so need to perform a large number of changes to distribute an existing application and most of these changes require thorough and extensive knowledge of both the application structure and the underlying system. Distributing an application involves splitting up the functionality of an application into independent/communicating entities so that they can execute in a distributed fashion while minimizing the interaction among the entities. For example, consider the scenario of a large simulation based application written for a centralized environment which collects some data from several sensors, runs some simulation on that data, analyzes and transforms the simulated results (if necessary) and then eventually visualizes the results. Currently if the programmer wishes to execute this application on a heterogeneous distributed system he/she needs to undertake a sequence of complicated activities to ensure efficient distributed execution of the application. Firstly, 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 which partitions should be given to which 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 high skill base that most programmers lack. Middleware paradigms such as Java RMI and CORBA are useful to a certain extent, but to utilize them, a programmers need advanced knowledge of complex programming interfaces and paradigms.

- **Resource Allocation:** Application performance can vary significantly depending on various resource assignments. When an application is deployed in 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 case of the above sensor, simulator, analysis and visualization example, perhaps the most effective way to distribute the application is to let the sensors run on moderately loaded and less powerful standalone nodes, simulation run on a high performance server and 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 so that the overall communication is minimized. However, identifying the resource requirements of an application and mapping the components to the resources is difficult and time consuming for application developers. 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, a self-managing approach should choose the right set of resources for each application and its components to satisfy the application's configuration and performance requirements, to avoid scarcity of resources, to accomplish certain goals, or to enforce certain user-defined policies or constraints (e.g. the simulator component

requires some input data that resides on Machine A, so the simulator code should execute on Machine A).

- **Heterogeneity**: The proposed infrastructure incorporates a large number of independent and distributed computational and communication resources where the challenge is to support processing at any node despite varying capabilities and environment.
- **Scalability**: A self-managed system should promote scalability. It should incorporate decentralized approaches which are essential to overcome high communication overheads and to enhance robustness.
- **Adaptability**: The proposed infrastructure is highly dynamic and is continuously evolving during the lifetime of an application. This includes the availability and the load on resources, changes in network conditions, etc. Application needs become dynamic when the interactions among runtime instances change during execution. To deal with these dynamic behaviors, it is crucial to support dynamic adaptation such as repartitioning, redeployment and reconfiguration. Incorporating dynamic behavior into an existing application such that it executes effectively in dynamic environments, not envisioned during the original design and development, is more challenging than developing new adaptable applications.
- **Uncertainty**: Uncertainty can be caused by the dynamic nature of the runtime environment and application, which may introduce unpredictable behavior and can only be detected and resolved during runtime. There are other factors that also may trigger uncertainty such as the failure of a system component, decisions motivated by partial or incomplete knowledge (typical in the case of decentralized architecture), etc.

1.2 Goals

The goal of our research is to build a self-managed distributed system that explores the "all care and no responsibility" principle of distribution whereby the average programmer does not wish to take responsibility for the physical distribution and coordination of the application but is, however, concerned with metrics such as the

application throughput and total execution time. Our aim is to provide an autonomic distributed system by automatically injecting self management and distribution capabilities into the user code so that the programmers do not have to deal with the distribution, management or optimization issues and, at the same time, system resources are utilized properly. The objective is to build a cost effective, self-managed, computing environment by offering spare compute cycles to the scientific and engineering research groups to satisfy their various demands including ease of use, performance, dynamism, adaptability, scalability, heterogeneity and so on and at the same time maximizing the use of existing resources. More specifically, the goals of our system are as follows:

- **Autonomic environment:** The objective is to provide an autonomic environment that transforms the existing centralized application in a way so that it becomes self-configured, self-optimized, self-healing and self-protecting.
- **Automatic Partitioning and Distribution:** ADE should be able to analyze an existing application, determine program partitions and their interactions automatically, distribute them while allocating appropriate resources, and coordinate them transparently at runtime. In general, ADE should be able to fully automate the partitioning and distribution tasks for the class of applications that interests us. However, in the proposed system, knowledgeable users can override system generated distribution and placement decisions by providing high-level policies and constraints.
- **Resource utilization:** ADE should work towards efficiently utilizing available resources. Resources are highly dynamic both in activity and availability. Therefore, the system must monitor itself and make the allocation and migration decisions based on the application characteristics and requirements as well as various runtime factors such as user demands, machine failure, network connectivity changes, workload, etc.
- **Efficiency:** ADE should optimize itself at both the individual user application level and as an execution environment. Instead of using a single optimization criterion such as minimum application response time, the proposed approach requires evaluating a certain utility function that combines various factors such as

response time, application priority, throughput, bandwidth utilization, etc. The aim is to dynamically maximize the utility function based on temporal factors.

1.3 Scope and Assumptions

Our goal is to design and implement an autonomic architecture to perform the distribution of user applications across the available resources and provide support for self-management. However, addressing all the issues related to self-management is beyond the capabilities of a single Ph.D. research project. ADE is a work-in-progress [9,10,11] currently being studied as a number of Ph.D. projects, and different aspects of self-management are addressed by different Ph.D. projects. The scope of this proposal is to explore two aspects of self-management: self-configuration and self-optimization. More specifically, this proposal is concerned with the identification of program partitions and the deployment and execution of those partitions across a self-organized network to maximize the overall utility of the system. The other two aspects of autonomic computing defined in [5,6] are the focus of other research projects [11,12,13] under ADE. The specific objectives that this proposal attempts to pursue are:

1. To automatically identify the communicating application components, and their dependencies, within a centralized application and to deploy them automatically to best take advantage of the underlying distributed computing resources.
2. To enable distributed resources to self-organize and self-manage.
3. To develop techniques that self-optimize the program execution and the use of distributed computational resources in order to maximize the system's overall business utility rather than focusing on single metrics such as minimizing execution time or maximizing throughput.

It is impractical to expect that all centralized applications can be distributed without rewriting the application's source code to achieve acceptable performance. ADE targets high performance scientific and engineering applications, where the computation-to-communication ratio is significant and the partitioning of tasks across the network results in increased performance. These applications typically make significant computation demands which are difficult to meet with a set of co-located computer

clusters. Adaptive and distributed execution environments make these computations easier and also create potential to improve performance via dynamic adaptation. More specifically, the proposed system targets two types of application as follows:

- *Concurrent execution application*: These applications are characterized by a large number of computationally intensive independent tasks, which makes it feasible to deploy them in a distributed environment despite the communication overhead. By load balancing and parallelizing the execution, the distributed version may result in some performance speedup. Examples of this type of application include SETI@home [14], factoring large numbers [15], genomics [16], etc.
- *Inherently distributed application*: These are the types of applications that must execute in a distributed setting to fulfill their functional constraints. Often these applications are sequential in execution; however the goal is to provide a distribution that meets the constraints with minimum communication overhead. For example, consider the sense-simulate-analyze-visualize application discussed in section 1.1. A large number of scientific and engineering applications can be characterized by this notion, where data is acquired by the sensor, then forwarded to a simulator for simulation, then passed to some analyzer that analyzes, operates and transforms the data, and eventually delivers it to some clients for visualization.

CHAPTER 2: Background and Related Works

ADE engages autonomic elements to automatically partition a centralized application and to deploy it over a set of distributed heterogeneous machines while maintaining the efficiency of the deployment. ADE addresses a few important issues such as self-management, automatic partitioning, resource utilization, decentralization and dynamic behavior in an integral way. As a result, ADE draws on many approaches and techniques practiced in research fields like Autonomic Computing, Automatic Program Partitioning and Distribution, Grid Computing and Adaptive Systems, and becomes relevant to the research projects in these areas. The following few sections detail the basics of these research fields along with the relevant research projects.

2.1 Autonomic Computing

The increasing complexity of computing systems, and their interaction 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 [17]. The term *autonomic* derives from the human body's autonomic nervous system, which controls key functions without conscious awareness or involvement [7]. An autonomic system, modeled after the autonomic nervous system, refers to a system that is able to protect itself, recover from faults, reconfigure as required by changes in the environment and always maintain its operations at a near optimal performance, all with minimal human intervention. Some benefits of autonomic computing include reduction of costs and errors, improvement of services and reduction of complexity [18].

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 converge these disciplines into a single field namely Autonomic Computing.

2.1.1 Characteristics of Autonomic Systems

An autonomic computing system can be a collection of autonomic components, which can manage their internal behaviors and relationships with others in accordance to high-level policies. The principles that govern all such systems is summarized by eight defining characteristics [18]:

- *Knowledge of itself*: An autonomic system must have detailed knowledge of its components, current status, capabilities, limits, boundaries, interdependencies with other systems, and available resources.
- *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 systems 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.

- *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.
- *Knowledge of environment and context*: An autonomic system must be aware of its execution environment and the context surrounding its activity, and be able to react to changes accordingly.
- *Ability to function in a heterogeneous computing world*: An autonomic system must be able to function in a heterogeneous world and consequently it must be built on standard and open protocols and interfaces. In other words, an autonomic system can not, by definition, be a proprietary solution.
- *Ability to Anticipate*: An Autonomic Computing System can anticipate its optimal required resources while hiding its complexity from the end user and attempts to satisfy user requests.

It should be noted that, among these eight properties, self-configuration, self-healing, self-optimization, and self-protection are considered as major characteristics and the rest are considered as minor characteristics.

2.1.2 Architectural Framework for Autonomic Systems

IBM researchers have established an architectural framework for autonomic systems [5]. 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. 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

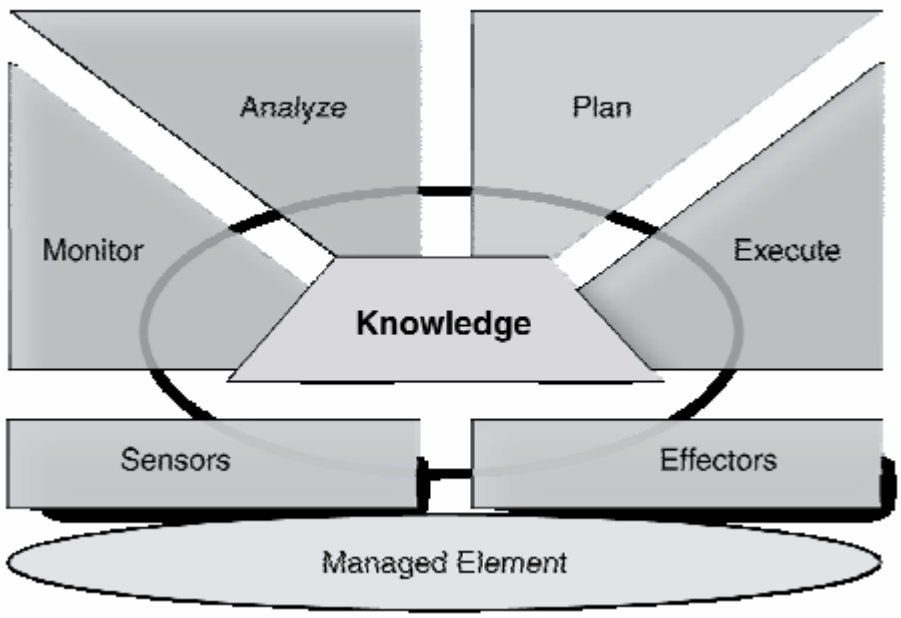

Figure 1 . Autonomic Element.

engine finally adjusts parameters of the managed element by means of effectors and stores the affected values in the knowledge base.

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.

2.1.3 Policies

Policy plays a fundamentally important role to play 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, 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 [19]. In this framework, a policy will 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, as follows:

- **Action Policies:** An action policy determines the action that should be taken when the system is in a given current state. Typically this action takes the form of "IF (condition) THEN (action)," where the condition specifies either a specific state or a set of possible states that all satisfy the given condition. This type of policy is generally necessary to ensure that the system is exhibiting rational behavior.
- **Goal Policies:** Rather than specifying exactly what to do in the current state, goal policies specify either a single desired state, or one or more criteria that characterize an entire set of desired states and let the system itself figure out how to achieve the desired state. The system generates rational behavior from the goal policy by using sophisticated planning or modeling algorithms. This type of policy permits greater flexibility and frees the human administrator from learning low-level details of system function.
- **Utility-function policies:** Utility function policies define an objective function that aims to model the behavior of the system at each possible state. Instead of performing a binary classification into desirable versus undesirable states, they ascribe a real-valued scalar desirability to each state. Since the most desired state is not specified in advance, it is 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. On the other hand, utility function policies can require policy authors to specify a multidimensional set of preferences, which may be

difficult to elicit; furthermore they require the use of modeling, optimization, and possibly other algorithms.

2.1.4 Related Work

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 similar to this study i.e. focus primarily on self configuration and self optimization.

Accord [20] is a component-based programming framework to support the development of autonomic application in grid environments. 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 [21] 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 [22] 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-configuring and self-healing by using a mobile agent system.

The aforementioned approaches [20,21,22] 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 the existing ones. On the contrary, ADE is 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 [23] 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, in Unity resource-level utility functions for multiple application environments are sent to an element called the Resource Arbiter, which computes 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 [24] 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 study 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 like 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 [25] 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 my research aims to model both the application and the underlying network into a common abstraction in order to achieve self-managed deployment. Also Rainbow is inherently centralized, with monitoring and adaptation performed within a single Rainbow instance.

Astrolabe [26] 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 [27,28] 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.

None of the abovementioned systems has the same goal as this research. Our objective is to provide self-managed deployment of centralized applications in distributed settings. More specifically, this proposal 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.

2.2 Automatic Application Partitioning and Distribution

Automatic partitioning systems are able to 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, mostly Java-based [29,32,33]. 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 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 few sections give a brief overview of some of these systems.

2.2.1 Related Work

Java Party [29] 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 way that the Runtime Manager runs as a central component, keeps 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 [30] 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 lot 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 byte-code.

Pangaea [31] 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 [32] is a Java byte-code 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 [33] is GUI-based and performs all transformations at the byte-code 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 with 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 [34] 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 application. 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 [35] automatically translates monolithic applications written in Java byte-code 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) [36, 37, 38, 39] 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 and therefore lose 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 ADE is that ADE's 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. ADE, 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 ADE compared to the above systems. Some systems rely on user's knowledge of the application to a certain extent and require the user to specify the network locations of hardware and software resources and annotate the code using them directly [30,31,32], while others completely

rely on the results of their automatic analysis and do not give the user any control at all [35]. However, in ADE, 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 ADE, user-level control can be exerted by providing high level policies and constraints.

2.3 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 [40].

2.3.1 Related Works

One particularly widely used technique for adaptation is aspect-oriented programming (AOP) [41], 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 [42] and ApectC# [43] are examples of systems that perform weaving at compile time; Weave.NET [44] and Aspect.NET [45] on the other hand perform weaving after compile time but before load time; and A Dynamic AOP-Engine for .NET [46], AspectIX [47] and CLAW [48] perform dynamic weaving at the byte-code 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 [49], Adaptive Java [50], and PCL [51] 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 the 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 [52], Dynamic TAO [53], Open ORB [54], QuO [55], Squirrel [56], and IRL [57]. 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 [58] and the CORBA Component Model (CCM) [59].

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 [60], metaXa [61], Guarana [62] and PROSE [63]. 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: RESEARCH PLAN

This chapter details the design and architecture of ADE and then elaborates how the main concern of this proposal - self configuration and self-optimization fit into the overall flow of operation. It then presents the three-tier architecture derived to realize the self-managed deployment of a centralized application in a distributed environment. The rest of the chapter then elaborates the research plan to accomplish different aspects of application deployment in ADE while justifying each choice.

3.1 Project ADE: Enabling Self-managed Distribution

ADE 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 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 the proposed system performs analysis and transformations at the byte-code level. However, applications for which source code is available can be transformed to byte-code and can exploit the benefits offered by this research.

3.1.1 Flow of Operation in ADE

Figure 2 shows ADE's overall flow of operation [11]. At first, a code analyzer statically inspects the user supplied byte-code to derive an object interaction graph. Based 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.

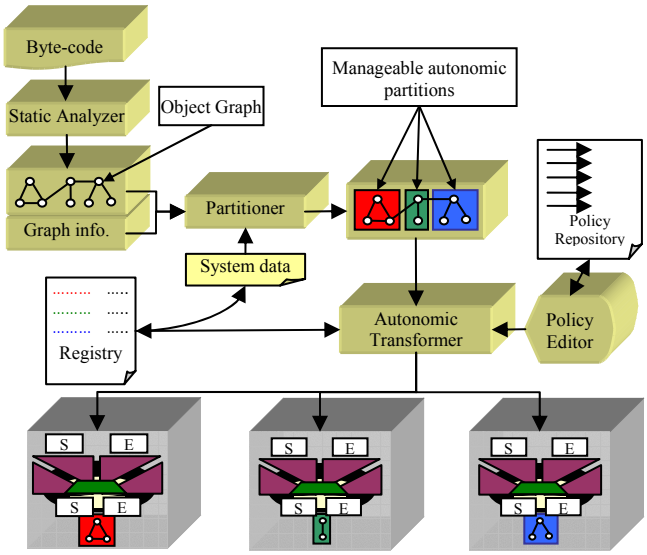

Figure 2. Overall flow of operation in ADE.

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. the 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 byte-code to byte-code 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.

3.1.2 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 run time 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 can change at runtime and so does the application's communication pattern, consequently the autonomic element needs to adapt accordingly. To achieve that, 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, etc.

3.2 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.

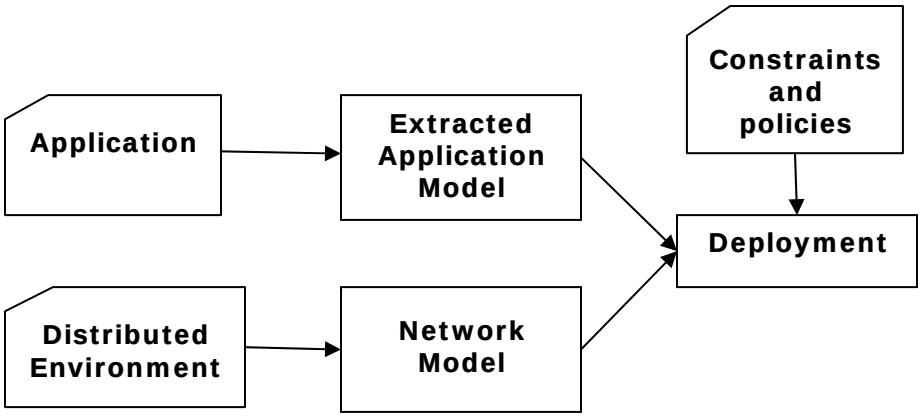

Figure 3. Application Deployment Process in ADE.

A three step process is designed to perform this mapping in ADE 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 and storage capabilities, workloads, etc. and then organizing them according to network proximity (considering latency, bandwidth, etc). 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, ADE 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.

Once the application components are automatically deployed into the distributed environment, dynamic reconfiguration can be triggered as needed. Based on the observations obtained by the monitoring of the system resources, ADE may automatically migrate application components, reallocate resources and redeploy the application graph. With respect to the self-optimizing criteria, ADE is designed to maximize a specific utility function [8] 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. During execution, resource allocation and other operating conditions may change; the corresponding change in the overall utility of the system can be calculated by this utility function and decisions can be taken toward maximizing this value.

3.3 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.
- 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 *Underlying layer* organizes the physical nodes in the network and is utilized by the upper level autonomic layer for deployment.

3.4 Application Layer

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 and the system needs detailed knowledge about it. An application can be

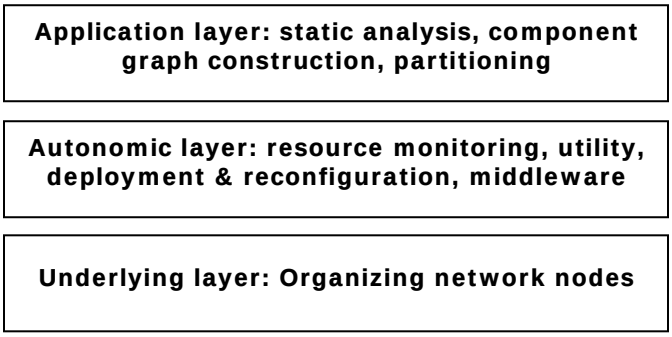

Figure 4. Deployment Architecture.

characterized by a set of components that communicate with each other in a certain way. In this proposal, an application is modeled as a graph consisting of application components and the interactions among them. Analyzing and representing software in terms of its components and their internal dependencies is important in order to provide the self managing capabilities because this is actually the system's view of the runtime structure of a program. Well structured graph-based modeling of an application 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 such an application graph, two pieces of information must be determined, namely: 1) the resources (i.e. computation time, storage, network and so on) consumed by each application component and 2) the dependencies (directionality and weight) among the components which is caused by the interactions among them. Therefore it is necessary to construct a node-weighted, edge-weighted directed graph $G = (V, E, w_g, c_g)$, where each node $v \in V$ 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 node v is $w_g(v)$ and represents the amount of computation that takes place at component v and the communication weight $c_g(u,v)$ captures the amount of communication (volume of data transferred) between nodes u and v. Figure 5 illustrates an example application graph with computational and communication weights. When deployed to 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.

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. There are several Java automatic partitioning tools [29, 30, 32, 33], however, they only detect interaction at the class level, and therefore perform partitioning at the class granularity and limit the opportunity to exploit object level

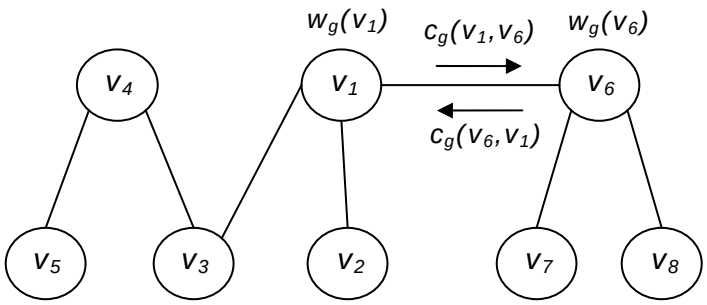

Figure 5. An application graph labeled with weights.

concurrency. To our knowledge, Spiegel's Pangaea [31] is the only system that performs analysis at the object level and potentially is able to fulfill our purpose. Spiegel's algorithm statically analyzes the Java source code and produces a directed graph where nodes and edges represent runtime instances and relations (*create*, *use* and *reference)* among them respectively.

While Spiegel's algorithm [31] provides important insight about object dependency graph construction, it is not sufficient for our purpose. For instance, the original algorithm simply produces a directed graph representing the runtime instances and their interactions. In contrast, ADE employs a weighted directed graph to effectively extract the computation and communication requirements of the objects and their dependencies. Moreover, instead of having a general *use* relation between a pair of objects, ADE's 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/write relations become valuable when determining distribution policies for objects. The original implementation assumes the presence of source code, while this study performs analysis at the byte-code level. Consequently, significantly different algorithmic aspects and implementation strategies need to be incorporated to fulfill the intents of this research.

This research plans to perform the static analysis of Java byte-code on top of the Jimple [64] representation, which is part of the Soot [64] framework. The Soot framework is a set of Java APIs for manipulating and optimizing Java byte-code. We plan to analyze the complete application, therefore by using Soot we first read all class files that are required by the application starting with the main method and recursively

loading all 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 byte-code. Jimple only involves 19 kinds of instruction and as a result is much easier to manipulate compared to stack oriented byte-code representation that involves 201 different instructions.

3.5 Underlying Layer

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. This research aims to organize the heterogeneous pool of 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 that correspond to low latency and high bandwidth between network nodes. The deployment of the application graph then can be performed in a utility-aware way, without having full knowledge about the underlying resources and without calculating the utility between all pairs of network nodes.

To achieve this, a hierarchical model of the computing environment is utilized where the execution begins at the root and each node either executes the tasks assigned to it or propagates them to the next level. More specifically, this proposal adopts a tree to model the underlying heterogeneous infrastructure. Each node in the tree has the sole responsibility for deciding whether to execute the components allocated to it or send them down the hierarchy. Each parent node is capable of calculating the utility of its child based on processor speed, workload, communication delay, bandwidth, etc. and selects its best child's subtree to delegate the tasks in a way such that the overall communication is minimized and the delegated component's resource requirements are satisfied.

There are three main advantages of representing the underlying layer as a tree. First, a tree model promotes decentralization and therefore is scalable. Each deployment decision can be made locally, which may not be optimal considering centralized deployment, but certainly is scalable and adaptive. 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. The proposed tree model can grow and reconfigure itself 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 redeploying the subtree assigned to it, based on the changes in utilities of its children. Optimizing certain utility functions globally is certainly more attractive, however it 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 promises to provide better adaptability and scalability in such a dynamic environment.

Third, the model fits very well with the classes of applications we are concerned with. For instance, concurrent execution applications basically exhibit master-slave behavior and easily correlate with the tree model. Inherently distributed applications on the other hand can be modeled as divide and conquer type applications where components are divided among partitions and allocated to the children for execution. 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.

The proposed hierarchical 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 is used to model the underlying resources, which is built on the fly on top of the existing network topology. Such an architecture was utilized recently in [65,66]. Figure 6 shows a small computing environment where resources are distributed in three domains and Figure 7 illustrates this environment hierarchically organized as a tree.

Formally, the entire network is represented as a weighted tree $T = (N, L, w_t, c_t)$, where N represents the set of computational nodes and L represents network links among them. The weights attached to the nodes and edges represent the associated computation and communication costs. The computational weight $w_t(n)$ indicates the cost associated with each unit of computation at node n. The communication weight $c_t(m,n)$ models the cost associated with each unit of communication of the link between node m and n considering both bandwidth and latency. Figure 7 shows an example tree model with computation and communication weights. When two nodes are not connected directly, their communication weight is the sum of the link weights on the shortest path among them. Therefore, larger values of node and edge weights translate to slower nodes and slower communication respectively.

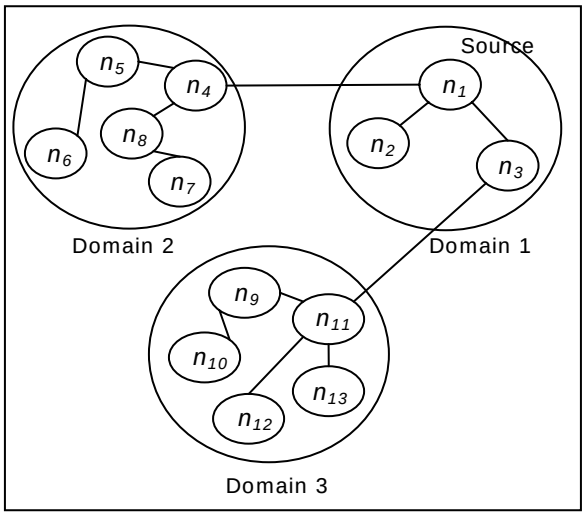

Figure 6. Sample target distributed environment spans at three domains.

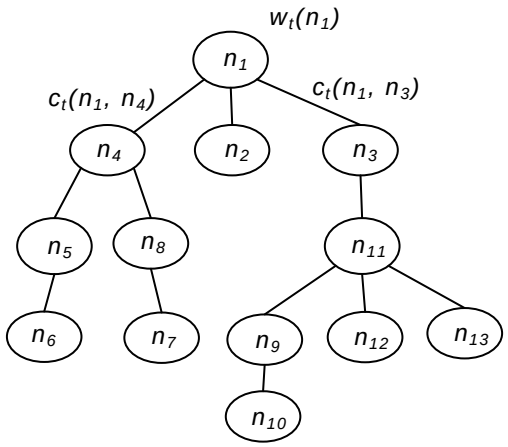

Figure 7. Hierarchical model of the environment.

3.6 Autonomic Layer

The purpose of the autonomic layer is to ensure self-organization and self-optimization. To achieve this, the autonomic layer is divided among the following modules.

3.6.1 Middleware Module

This module is responsible for providing middleware services such as handling communication among distributed entities.

3.6.2 Resource Monitoring Module

The purpose of this module is to manage resource availability information across the network. Each resource 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 resource. The performance attributes are measurable entities, e.g. for a computational resource, the attributes may include processor speed, cache size, memory size, memory latency, current load, memory usage, etc. Based on this information, the system's overall utility is recalculated and the application graph is redeployed in response to the changes in resource conditions. This research intends to utilize directory services to provide information about different resource instances in the environment. The chosen directory service should provide a simple interface, easy and efficient enough to insert and retrieve resource instances, should operate in highly dynamic environments and most importantly should scale well in case of a large number of instances. To reduce communication overhead, the intention is to initiate resource updates only when the environment changes instead of costly periodical updates.

3.6.3 Utility Evaluation Module

In a distributed environment, the utility can be calculated based on many criteria such as application performance, resource utilizations, user defined policies, or economic concerns. These issues can be associated with different objective functions of the optimization problem. In this research, the utility function governs both the initial placement of application components and their reconfigurations. The goal is to minimize

the average application execution time and to provide high utilization ensuring that both application level (each application may have a different importance to the system) and system level requirements and constraints are satisfied. Toward our goal, we have identified that the following criteria should be considered by the utility function:

- While mapping partitions containing a large number of components in the tree network, nodes with a higher degree of connectivity should result in a higher utility as a higher degree allows more directions for partition growth.
- Nodes with a faster communication link should be preferred over nodes with a slower communication link.
- Faster and less busy nodes should be favored over slower and overloaded nodes.
- High priority applications should be preferred during deployment over low priority jobs.

3.6.4 Initial Deployment Module

Once both the application and underlying resources have been modeled, the deployment problem reduces to mapping 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. In ADE, the assumption is that the application can be submitted to any node which acts as the source of the application and can terminate either at the source or at one or more clients at different destination nodes. The nodes in the system are self-organized into a tree-shaped computing environment rooted at the source of execution according to the model described in Section 3.5

Once the application graph G is submitted to the root node, the root then decides which application components to execute itself and which components to forward to its child's sub-tree so that the overall mapping results in the highest utility. 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 [67] can be exploited to collapse several application components to 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.

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 decides to execute itself) and $v_k \in V_s$ (represents the set of components that belong to a partition that a parent decides 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. We expect to derive an appropriate utility function that will take into consideration all the factors detailed in section 3.6.3 and will govern the initial deployment efficiently.

3.6.5 Self Optimization Module

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. Reconfiguration can be 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 within a subtree is expected to be a less expensive process because of the way the underlying network is modeled. Through the resource monitoring module, each parent node periodically measures the workload at each child and its bandwidth to the child and consequently changes computational and communication weights attached to that child. By incorporating this monitored information into the utility function, it is possible for a parent to observe the change in utility due to the changes in network and compute nodes, and therefore reconfiguration can initiate autonomously. Reconfiguration is costly and disruptive, therefore, it is not feasible to initiate reconfiguration unless it is productive. This research plans to trigger reconfiguration whenever the utility drops more than a

certain threshold (user specified or system generated by comparing the utility during initial deployment).

3.7 Evaluation

We intend to evaluate the performance of the self-managed deployment architecture by proof-of-concept implementation and by using simulation and experiments. More specifically we want to evaluate the effectiveness of self-configuration and self-optimization of the application graph in the proposed self-organized tree network. Since, our aim is not to use a single application or system specific attribute such as response time, throughput or resource utilization for evaluation; we will be using system's overall business utility instead as the performance metric in our experiments. For instance, we will be justifying our architecture by measuring the utility achieved by it and will compare it against the utility produced by other models. The experiments are outlined as follows.

- In order to justify the applicability of our self-organized tree network, we will conduct some experiments that will evaluate the quality of the utility-driven configuration of an application graph which is automatically generated by our approach. For comparison, we will manually create a good initial configuration with prior knowledge of system parameters. We will then find the configuration generated by our approach and will examine the deviation of it from the manually generated one. We will also measure the time needed for the computation to reach all nodes of the tree and the overall execution time of an application.

- In order to justify the effectiveness of our decentralized utility evaluation, we will conduct some experiments that compare the overall utility of an initially deployed application graph using the centralized model (a central node keeps track of utility information for all other nodes and based on this global knowledge calculates the optimal utility) as opposed to the decentralized utility calculation in our tree network. We will also estimate the cost associated with both approaches to determine the cost/optimality tradeoff.

- To examine the effectiveness of our dynamic reconfiguration, we will perform experiments that show the variation of utility for a deployed application graph with changing network conditions. With this set of experiments, we expect to find out how costly the reconfiguration can be and whether it is worth considering or not. With rigorous experiments, we expect to find a suitable threshold value for triggering reconfiguration.

- Our next set of experiments will study the behavior of the deployment architecture in case of more than one application graph. The intention is to observe how ADE behaves in case of simultaneous deployment of many applications. We will also simulate the factors that trigger self-optimization such as network traffic, increased processor workload, reduced bandwidth, arrival of a higher priority job, etc. to see how those changes effect the utility evaluation and how the system optimizes itself.

3.7 Expected Contributions of This Research

As systems and application grow in scale and complexity, attaining the desired level of performance in today's highly dynamic and uncertain environment using current approaches based on global knowledge, centralized scheduling and manual reallocation becomes infeasible. To address this challenge, application components need to be self-managed in order to be able to dynamically detect and respond, quickly and correctly, to changes in the application's behaviors and the state of the underlying environment.

In my thesis, I expect to design and implement algorithms and mechanisms for achieving self-managed deployment of high performance scientific and engineering applications in highly dynamic and unpredictable distributed environment. My approach is to incorporate autonomic entities to handle the complexities associated with distribution, configuration, coordination and efficient execution of application components and to adapt to the changes in application behavior and the underlying environment. I believe that such an automated approach results in higher utilization of distributed resources while meeting application performance demands. In particular, I am expecting to make following key contributions towards the design of self-managed deployment.

- I intend to develop techniques for automatically identifying application components and their estimated resource requirements within a centralized application and use them in order to model the application into a graph abstraction.
- I expect to develop techniques that allow the distributed resources to self-organize in a utility-aware way while assuming minimal knowledge about the underlying environment.
- To achieve efficient initial deployment of application components to the network nodes and their runtime reconfigurations, I plan to design a decentralized scheduling algorithm based on utility functions. The approach is novel in its consideration of overall system's utility (combining multiple application and system level attributes) during deployment and reconfiguration. The approach is scalable and adaptive in its use of 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.

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 Research. *Autonomic Computing.* http://www.research.ibm.com/autonomic
8. W. E. Walsh, G. Tesauro, J. O. Kephart, R. Das. Utility functions in autonomic systems. *Proceedings. Of the First International Conference on Autonomic Computing (ICAC),* pages 70-77, 2004.
9. M. J. Oudshoorn, M. M. Fuad and D. Deb. Towards an Automatic Distribution System - Issues and Challenges. *PDCN*, pages 399-404, 2005.
10. M. M. Fuad and M. J. Oudshoorn. An Autonomic Architecture for Legacy Systems, *Third IEEE Workshop on Engineering of Autonomic Systems (EASe)*, pages 79-88, 2006.
11. M. M. Fuad. An Autonomic Software Architecture for Distributed Applications. *Ph. D. thesis*, Department of Computer Science, Montana State University, USA, 2007.
12. M. M. Fuad and M. J. Oudshoorn. Transformation of Existing Programs into Autonomic and Self-healing Entities. *The 14th IEEE International Conference on the Engineering of Computer Based Systems (IEEE/ECBS)*, pages 133-144, 2007.

13. M. M. Fuad, D. Deb and M. J. Oudshoorn, An Autonomic Element Design for a Distributed Object System, *ISCA 20th International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 273-279, 2007.
14. SETI@home. http://setiathome.ssl.berkeley.edu, 2001.
15. 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*, pages 382–394, 1996, Vol. 1163 of LNCS.
16. M. Waterman and T. Smith. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, Vol. 147, pages195–197, 1981.
17. S. Hariri , B. Khargharia , H. Chen , J. Yang , Y. Zhang , M. Parashar, H. Liu. The Autonomic Computing Paradigm, *Cluster Computing*, Vol. 9 No.1, pages 5-17, 2006.
18. R. Murch. Autonomic Computing. *Prentice-Hall*, 2004.
19. J. O. Kephart and W. E. Walsh. An Artificial Intelligence Perspective on Autonomic Computing Policies. In *International Workshop on Policies for Distributed Systems and Networks,* pages 3–12, 2004.
20. 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, pages. 341 – 352, 2006.
21. 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.
22. X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. AUTONOMIA: An Autonomic Computing Environment. *Proc. of the 2003 IEEE International Performance, Computing, and Communication Conference*, pages 61-68, 2003.
23. D. M. Chess, A. Segal, I. Whalley and S. R. White. Unity: Experiences with a Prototype Autonomic Computing System. *1st International. Conference on Autonomic Computing (ICAC)*, pages 140-147, 2004.

24. 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.
25. S. Cheng, A. Huang, D. Garlan, B. Schmerl, P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *First International Conference on Autonomic Computing (ICAC)* pages 276-277, 2004.
26. 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.
27. K. Schwan et al. Autoflow: Autonomic information flows for critical information systems. Autonomic Computing: Concepts, Infrastructure, and Applications, CRC Press, 2006.
28. V. Kumar, B. F. Cooper, K. Schwan. Distributed Stream Management using Utility-Driven Self-Adaptive Middleware. *Second International Conference on Autonomic Computing (ICAC)*, pages 3-14, 2005.
29. M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, Vol. 9, No. 11, pages 1125-1242, 1997.
30. M. Dahm. Doorastha—a step towards distribution transparency. *JIT,* 2000.
31. A. Spiegel. Automatic Distribution of Object-Oriented Programs. *PhD thesis*, Fachbereich Mathematik u. Informatik, Freie Universitat, Berlin, 2002.
32. M. Tatsubori, T. Sasaki, S. Chiba and K. Itano. A Byte-code Translator for Distributed Execution of Legacy Java Software. *ECOOP*, pages 236-255, 2001.
33. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. *ECOOP*, 2002.
34. G. C. Hunt, and M. L. Scott. The Coign Automatic Distributed Partitioning System. *3rd Symposium on Operating System Design and Implementation (OSDI)*, pages 187-200, 1999.
35. R. E. Diaconescu, L. Wang, Z. Mouri and M. Chu. A Compiler and Runtime Infrastructure for Automatic Program Distribution. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

36. Y. Aridor, M. Factor, and A. Teperman. CJVM: a Single System Image of a JVM on a Cluster. *ICPP*, 1999.
37. 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 Trans. on Computer Systems*, Vol. 16, No.1, pages 1-40, 1999
38. J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. *13th ACM Symposium on Operating Systems Principles*, pages 152-164, 1991.
39. W. Yu, and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, Vol. 9, No. 11, pages 1213-1224, 1997.
40. 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.
41. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 1241, 1997.
42. G. Kiczales et al.. An Overview of AspectJ. In *Proceedings of European Conference on Object-Object Programming (ECOOP)*, pages 327–353, 2001.
43. H. Kim. AspectC#: An AOSD implementation for C#, *Masters Thesis*, Department of Computer Science, Trinity College, Dublin, September 2002.
44. 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.
45. B. Rasmussen et al. Aspect.NET - A Cross-Language Aspect Weaver. Department of Computer Science, Trinity College, Dublin, 2002.
46. A. Frei et al., A Dynamic AOP-Engine for .NET, Technical Report 445, Department of Computer Science, ETH Zurich, May 2004.
47. 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.

48. J. Lam. CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime Demonstration at the *1st International Conference on Aspect-Oriented Software Development*, 2002.
49. 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.
50. E. P. Kasten and P. K. McKinley. Adaptive Java: Refractive and transmutative support for adaptive software. Tech. Rep. MSU-CSE-01-30, Computer Science and Engineering, Michigan State University, 2001
51. 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.
52. 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.
53. F. Kon, M. Rom´an, 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.
54. 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.
55. 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.
56. 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.

57. R. Baldoni, C. Marchetti, and A. Termini. Active software replication through a three-tier approach. in *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, pages 109–118, 2002.
58. Sun Microsystems, http://java.sun.com/products/ejb/, Enterprise JavaBeans Technology, 2001.
59. Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/99-10-05, CORBA Components Model - FTF drafts for MOF chapter.
60. B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. in *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.
61. 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.
62. A. Oliva and L. E. Buzato. The implementation of Guaran´a on Java. Tech. Rep. IC-98-32, Universidade Estadual de Campinas, Sept. 1998.
63. A. Popovici, T. Gross, and G. Alonso. Dynamic homogenous AOP with PROSE. Tech. Rep., Department of Computer Science, Federal Institute of Technology, Zurich, 2001.
64. Sable research group, www.sable.mcgill.ca/soot.
65. O. Beaumont, A. Legrand, Y. Robert, L. Carter, J. Ferrante. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.
66. B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante, "Autonomous protocols for bandwidth-centric scheduling of independent-task applications", *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.
67. 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.