A FRAMEWORK FOR THE DYNAMIC RECONFIGURATION OF SCIENTIFIC APPLICATIONS IN GRID ENVIRONMENTS

By

Kaoutar El Maghraoui

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

_____
Dr. Carlos A. Varela, Thesis Adviser

_____
Dr. Joseph E. Flaherty, Member

_____
Dr. Ian Foster, Member

_____
Dr. Franklin Luk, Member

_____
Dr. Boleslaw K. Szymanski, Member

_____
Dr. James D. Teresco, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2007
(For Graduation May 2007)

A FRAMEWORK FOR THE DYNAMIC RECONFIGURATION OF SCIENTIFIC APPLICATIONS IN GRID ENVIRONMENTS

By

Kaoutar El Maghraoui

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Dr. Carlos A. Varela, Thesis Adviser
Dr. Joseph E. Flaherty, Member
Dr. Ian Foster, Member
Dr. Franklin Luk, Member
Dr. Boleslaw K. Szymanski, Member
Dr. James D. Teresco, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2007
(For Graduation May 2007)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

It is with great pleasure that I wish to acknowledge several people that have helped me tremendously during the difficult, challenging, yet rewarding and exciting path towards a Ph.D. Without their help and support, none of this work could have been possible.

First and foremost, I am greatly indebted to my advisor Dr. Carlos A. Varela for his guidance, encouragement, motivation, and continued support throughout my academic years at RPI. Carlos has allowed me to pursue my research interests with sufficient freedom, while always being there to guide me. Working with him has been one of the most rewarding experiences of my professional life.

I am also deeply indebted to Professor Boleslaw K. Szymanski, my committee member, for supporting my work. He has been very instrumental to the realization of this work through his keen guidance and encouragement. Working with him has been a great pleasure. I am also very thankful to the rest of my committee members, Dr. Joseph E. Flaherty, Dr. Ian Foster, Dr. Franklin Luk, and Dr. James D. Teresco. I am grateful to them for agreeing to serve on my committee and for their valuable suggestions and comments.

Special thanks go to my colleague, Travis Desell for his key contributions to the design and development of the Internet Operating System middleware. Many thanks go also to the rest of the Worldwide Computing laboratory members, Wei-Jen Wang, Jason LaPorte, Jiao Tao, and Brian Boodman for their valuable comments and constructive criticism. My fruitful discussions and interactions with them helped me grow professionally.

I am grateful to the administrative staff of the Computer Science department, who have spared no efforts helping me in various aspects of my academic life at RPI. They were some of the best people I have ever worked with. In particular, I would like to thank Pamela Paslow for her constant help and for also being a true friend. She was always there for me in easy and difficult times. She has kept me on top of all the necessary paperwork. I would like also to thank Jacky Carley, Shannon Carrothers, Chris Coonrad, and Steven Lindsey.

I have been fortunate to have met great friends throughout my Ph.D journey. They have bestowed so much love on me. I am forever grateful for their moral support, encouragement, and true friendship. In particular I would like to thank Bouchra Bouqata, Houda Lamehamedi, Fatima Boussetta, and Khadija Omo-meriem. Special thanks go to Rebiha Chekired for caring for my baby, Zayneb, during the last year of my Ph.D. She acted as a loving and caring second mother to my baby during times I could not be around. I am forever grateful to her.

Last but not least, I am forever indebted to my husband, Bouchaib Cherif, my parents, my sisters Hajar and Meriem, my brother Ahmed, and the rest of my family. My husband has been a great source of inspiration to me. None of this would have been possible without his love, support, and continuous encouragement. My parents' prayers have always accompanied me. Their love keeps me going. My daughter Zayneb has been the greatest source of motivation and inspiration during the last year of my Ph.D. I am very lucky to have been blessed with her. I am grateful to all of them. This work is dedicated to my family.

ABSTRACT

Advances in hardware technologies are constantly pushing the limits of processing, networking, and storage resources, yet there are always applications whose computational demands exceed even the fastest technologies available. It has become critical to look into ways to efficiently aggregate distributed resources to benefit a single application. Achieving this vision requires the ability to run applications on dynamic and heterogeneous environments such as grids and shared clusters. New challenges emerge in such environments, where performance variability is the rule and not the exception, and where the availability of the resources can change anytime. Therefore, applications require the ability to dynamically reconfigure to adjust to the dynamics of the underlying resources.

To realize this vision, we have developed the Internet Operating System (IOS), a framework for middleware-driven application reconfiguration in dynamic execution environments. Its goal is to provide high performance to individual applications in dynamic settings and to provide the necessary tools to facilitate the way in which scientific and engineering applications interact with dynamic environments and reconfigure themselves as needed. Reconfiguration in IOS is triggered by a set of decentralized agents that form a virtual network topology. IOS is built modularly to allow the use of different algorithms for agents' coordination, resource profiling, and reconfiguration. IOS exposes generic APIs to high-level applications to allow for interoperability with a wide range of applications. We investigated two representative virtual topologies for inter-agent coordination: a *peer-to-peer* and a *cluster-to-cluster* topology. As opposed to existing approaches, where application reconfiguration has

mainly been done at a coarse granularity (e.g., application-level), IOS focuses on migration at a fine granularity (e.g., process-level) and introduces a novel reconfiguration paradigm, *malleability*, to dynamically change the granularity of an application's entities. Combining migration and malleability enables more effective, flexible, and scalable reconfiguration.

IOS has been used to reconfigure actor-oriented applications implemented using the SALSA programming language and iterative process-oriented applications that follow the Message Passing Interface (MPI) model. To benefit from IOS reconfiguration capabilities, applications need to be amenable to entity migration or malleability. This issue has been addressed in iterative MPI applications by designing and building a library for process checkpointing, migration, and malleability (PCM) and integrating it with IOS. Performance results show that adaptive middleware can be an effective approach to reconfiguring distributed applications with various ratios of communication to computation in order to improve their performance, and more effectively utilize dynamic resources. We have measured the middleware overhead in static environments demonstrating that it is less than 7% on average, yet reconfiguration on dynamic environments can lead to significant improvement in application's execution time. Performance results also show that taking into consideration the application's communication topology in the reconfiguration decision improves throughput by almost an order of magnitude in benchmark applications with sparse inter-process connectivity.

CHAPTER 1

Introduction

Computing environments have evolved from single-user environments, to Massively Parallel Processors (MPPs), to clusters of workstations, to distributed systems, and recently to *grid computing systems*. Every transition has been a revolution, allowing scientists and engineers to solve complex problems and sophisticated applications that could not be solved before. However every transition has brought along new challenges, new problems, and also the need for technical innovations.

The evolution of computing systems has led to the current situation, where millions of machines are interconnected via the Internet with various hardware and software configurations, capabilities, connection topologies, access policies, etc. The formidable mix of hardware and software resources in the Internet has fueled researchers' interest to start investigating novel ways to exploit this abundant pool of resources in an economic and efficient manner and also to aggregate these distributed resources to benefit a single application. Grid computing has emerged as an ambitious research area to address the problem of efficiently using multi-institutional pools of resources. Its goal is to allow coordinated and collaborative resource sharing and problem solving across several institutions to solve large scientific problems that could not be easily solved within the boundaries of a single institution. The concept of a *computational grid* first appeared in the mid-1990's, proposed as an infrastructure for advanced science and engineering. This concept has evolved extensively since then and has encompassed a wide range of applications in both the scientific and commercial fields [46]. Computing power is expected to become in the future a pur-

chasable commodity, like electrical power. Hence, the analogy often made between the electrical power grid and the conceptual *computational grid*.

1.1 Motivation and Research Challenges

New challenges emerge in grid environments, where the computational, storage, and network resources are inherently heterogeneous, often shared, and have a highly dynamic nature. Consequently, observed application performance can vary widely and in unexpected ways. This renders the maintenance of a desired level of application performance a hard problem. Adapting applications to the changing behavior of the underlying resources becomes critical to the creation of robust grid applications. Dynamic application reconfiguration is a mechanism to realize this goal.

We denote by an application's *entity*, a self-contained part of a distributed or parallel application that is running in a given runtime system. Examples include processes in case of parallel applications, software agents, web services, virtual machines, or actors in case of actor-based applications. Application's entities could be running in the same runtime environment or on different distributed runtime environments connected through the network. They could be tightly coupled, exchanging a lot of messages or loosely coupled, with no or few messages exchanged. Dynamic reconfiguration implies the ability to modify the mapping between application's entities and physical resources and/or modify the granularity of the application's entities while the application continues to run without any disruption of service. Applications should be able to scale up to exploit more resources as they become available or gracefully shrink down as some resources leave or experience failures. It is impractical to expect application developers to handle reconfiguration issues given the sheer size of grid environments and the highly dynamic nature of the resources. Adopting a middleware-driven approach is imperative to achieving efficient deployment of applications in a dynamic grid setting.

Application adaptation has been addressed in previous work in a fairly ad-hoc manner. Usually the code that deals with adaptation is embedded within the application or within libraries that are highly application-model dependent. Most of these strategies require having a good knowledge of the application model and a good

knowledge of the execution environments. While these strategies may work for dedicated and fairly static environments, they do not scale up to grid environments that exhibit larger degrees of heterogeneity, dynamic behavior, and a much larger number of resources. Recent work has addressed adaptive execution in grids. Most of the mechanisms proposed have adopted the application *stop and restart* mechanism; i.e., the entire application is stopped, checkpointed, migrated, and restarted in another hardware configuration (e.g., see [72, 110]). Although this strategy can result in improved performance in some scenarios, more effective adaptivity can be achieved if migration is supported at a finer granularity.

1.1.1 Mobility and Malleability for Fine-grained Reconfiguration

Reconfigurable distributed applications can opportunistically react to the dynamics of their execution environment by migrating data and computation away from unresponsive or slow resources, or into newly available resources. Application stop-and-restart can be thought of as application mobility. However, application entity mobility allows applications to be reconfigured with finer granularity. Migrating entities can thus be easier and more concurrent than migrating a full application. Additionally, concurrent entity migration is less intrusive.

To illustrate the usefulness of such dynamic application entity mobility, consider an iterative application computing heat diffusion over a two-dimensional surface. At each iteration, each cell recomputes its value by applying a function of its current value and its neighbors' values. Therefore, processors need to synchronize at every iteration with their neighbors before they can proceed on to a subsequent iteration. Consequently, the simulation runs as slow as the slowest processor in the distributed computation, assuming a uniform distribution of data. Clearly, data distribution plays an important role in the efficiency of the simulation. Unfortunately, in shared environments, the load of involved processors is unpredictable, fluctuating as new jobs enter the system or old jobs complete.

We evaluated the execution time of the heat simulation with and without the capability of application reconfiguration under a dynamic run-time environment: the application was run on a cluster and soon after the application started, artificial load

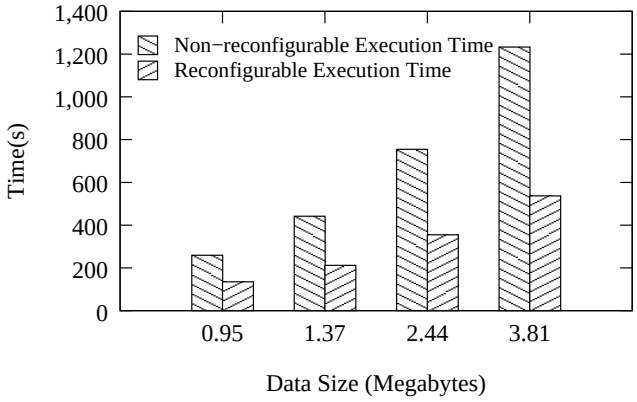

Figure 1.1: Execution time without and with autonomous migration in a dynamic run-time environment.

was introduced in one of the cluster machines. Figure 1.1 shows the speedup obtained by migrating the slowest process to an available node in a different cluster.

While entity migration can provide significant benefits in performance to distributed applications over shared and dynamic environments, it is limited by the granularity of the application's entities . To illustrate this limitation, we use another iterative application. This application is run on a dynamic cluster consisting of five processors (see Figure 1.2). In order to use all the processors, at least one entity per processor is required. When a processor becomes unavailable, the entity on that processor can migrate to a different processor. With five entities, regardless of how migration is done, there will be imbalance of work on the processors, so each iteration needs to wait for the pair of entities running slower because they share a processor. In the example, 5 entities running on 4 processors was 45% slower than 4 entities running on 4 processors, with otherwise identical parameters. One alternative to fix this load imbalance is to increase the number of entities to enable a more even distribution of entities no matter how many processors are available. In the example of Figure 1.2, 60 entities were used since 60 is divisible by 5, 4, 3, 2 and 1. Unfortunately, the increased number of entities introduces overhead which causes the application to run slower, approximately 7.6% slower. Additionally, this approach is not scalable, as the number of entities required for this scheme is the least common multiple of different combinations of processor availability. In many cases, the availability of

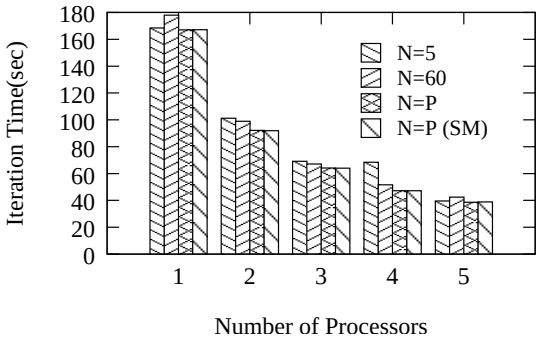

Figure 1.2: Execution time with different entity granularities in a static run-time environment.

resources is unknown at the application's startup so an effective number of entities cannot be statically determined. Figure 1.2 shows these two approaches compared to a good distribution of work, one entity per processor. N is the number of entities and P is the number of processors. N=P with split and merge (SM) capabilities uses entities with various granularities, while N=P shows the optimal configuration for this example (with no dynamic reconfiguration and middleware overhead). N=60 and N=5 show the best configuration possible using migration with a fixed number of entities. In this example, if a fixed number of entities is used, averaged over all processor configurations, using five entities is 13.2% slower, and using sixty entities is 7.6% slower.

To illustrate further how process's granularity impacts the node-level performance, we run an iterative application with different numbers of processes on the same dedicated node. The larger the number of processes, the smaller the data granularity of each process. Figure 1.3 shows an experiment where the parallelism of an iterative application was varied on a dual-processor node. In this example, having one process per processor did not give the best performance, but increasing the parallelism beyond a certain point also introduces a performance penalty.

We introduce the concept of *mobile malleable* entities to solve the problem of appropriately using resources in the face of a dynamic execution environment where the available resources may not be known. Instead of having a static number of entities, malleable entities can *split*, creating more entities, and *merge*, reducing the

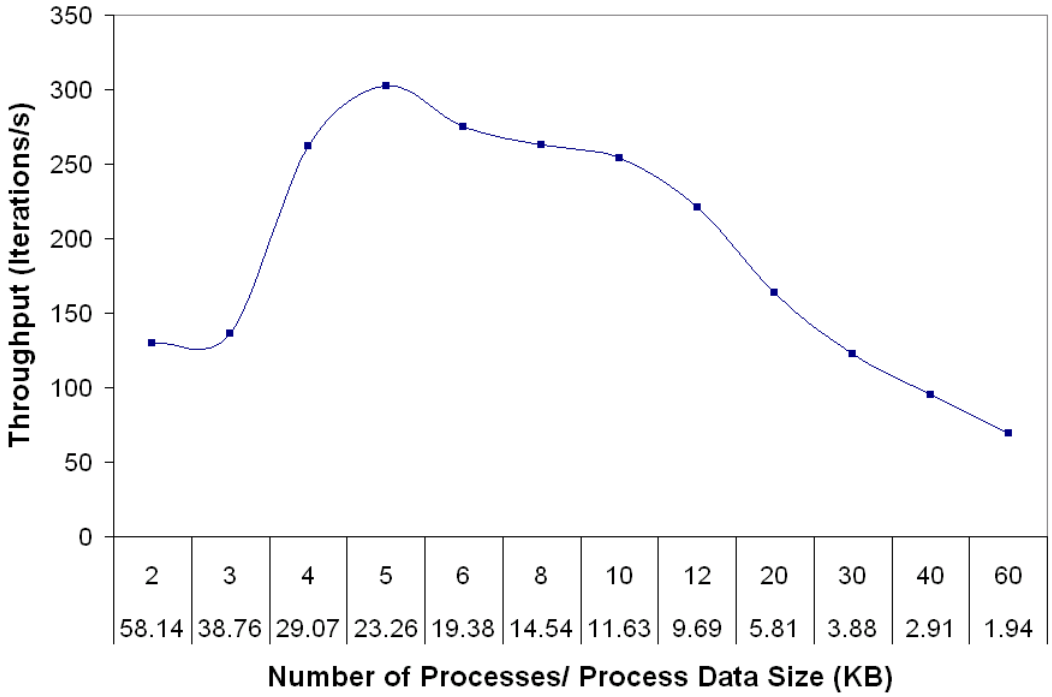

Figure 1.3: Throughput as the process data granularity decreases on a dedicated node.

number of entities, redistributing data based on these operations. With malleable entities, the application's granularity, and as a consequence, the number of entities, can also be changed dynamically. Applications define *how* entities split and merge, while the middleware determines *when* based on resource availability information, and *what* entities to split or merge depending on their communication topologies. As the dynamic environment of an application changes, in response, the granularity and data distribution of that application can be changed to utilize its environment most efficiently.

1.1.2 Middleware-driven Dynamic Application Reconfiguration

There are a number of challenges to enable dynamic reconfiguration in distributed applications. We divide them into middleware challenges, programming technology challenges, and the interface between the middleware and the applications.

Middleware-level Challenges

Middleware challenges include the need for continuous and non-intrusive profiling of the run-time environment resources and to determine when it is expected that an application reconfiguration will lead to performance improvements or better resource utilization. A middleware layer needs to accomplish this in a decentralized way, so as to be scalable. The meta-level information that the middleware manages must include information on the communication topology of the application entities to co-locate those that communicate extensively whenever possible, avoiding high-latency communication. A good compromise must also be found between highly accurate meta-level information—but potentially very expensive to obtain and with a cost of intrusiveness to running applications—and partial, inaccurate meta-level information—that may be cheap to obtain in non-intrusive ways but may lead to far from optimal reconfiguration decisions. Since no single policy fits all, modularity is needed to be able to plug in and fine tune different resource profiling and management policies embedded in the middleware.

Application-level Challenges

The middleware can only trigger reconfiguration requests to applications that support them. Programming models advocating for a clean encapsulation of state inside the application entities and asynchronous communication among entities, make the process of reconfiguring applications dynamically much more manageable. This is because there is no need for replicated shared memory consistency protocols and preservation of method invocation stacks upon entity migration. While entity mobility is relatively easy and transparent for these programming models, entity malleability requires more cooperation from the application developers as it is highly application-dependent. For programming models where shared memory or synchronous communication are used, application programming interfaces need to be defined to enable developers to specify how application entity mobility and malleability are supported by specific applications. These models make the reconfiguration process less transparent and sometimes limit the applicability of the approach to specific classes of applications, e.g., massively parallel or iterative applications.

Cross-cutting Challenges

Finally, applications need to collaborate with the middleware layer by exporting meta-level information on entity interconnectivity and resource usage and by providing operations to support potential reconfiguration requests from the middleware layer. This interface needs to be as generic as possible to accommodate a wide variety of programming models and application classes.

1.2 Problem Statement and Methodology

The focus of this research is to build a modular framework for middleware-driven application reconfiguration in dynamic execution environments such as Grids and shared clusters. The main objectives of this framework are to provide high performance to individual applications in dynamic settings and to provide the necessary tools to facilitate the way in which scientific and engineering applications interact with dynamic execution environments and reconfigure themselves as needed. Hence, such applications will be allowed to benefit from these rapidly evolving systems and from the wide spectrum of resources available in them.

This research addresses most of the issues described in the previous section through the following methodology.

A Modular Middleware for Adaptive Execution

The Internet Operating System (IOS) is a middleware framework that has been built with the goal of addressing the problem of reconfiguring long running applications in large-scale dynamic settings. Our approach to dynamic reconfiguration is twofold. On the one hand, the middleware layer is responsible for resource discovery, monitoring of application-level and resource-level performance, and deciding when, what, and where to reconfigure applications. On the other hand, the application layer is responsible for dealing with the operational issues of migration and malleability and the profiling of application communication and computational patterns. IOS is built with modularity in mind to allow the use of different modules for agents coordination, resource profiling and reconfiguration algorithms in a plug and play fashion. This feature is very important since there is no "one size fits all" method for

performing reconfiguration for a wide range of applications and in highly heterogeneous and dynamic environments. IOS is implemented in Java and SALSA [114], an actor-oriented programming language. IOS agents leverage the autonomous nature of the actor model and use several coordination constructs and asynchronous message passing provided by the SALSA language.

Decentralized Coordination of Middleware Agents

IOS embodies resource profiling and reconfiguration decisions into software agents. IOS agents are capable of organizing themselves into various virtual topologies. Decentralized coordination is used to allow for scalable reconfiguration. This research investigates two representative virtual topologies for inter-agent coordination: a peer-to-peer and a cluster-to-cluster coordination topology [73]. The coordination topology of IOS agents has a great impact on how quickly reconfiguration decisions are made. In a more structured environment such as a grid of homogeneous clusters, a hierarchical topology generally performs better than a purely peer-to-peer topology [73].

Generic Interfaces for Portable Interoperability with Applications

IOS exposes several APIs for profiling applications' communication patterns and for triggering reconfiguration actions such as migration and split and merge. These interfaces shield many of the intrinsic details of reconfiguration from application developers and provide a unified and clean way of interaction between applications and the middleware. Any application or programming model that implements IOS generic interfaces becomes reconfigurable.

A Generic Process Checkpointing, Migration and Malleability Scheme for Message Passing Applications

Migration or malleability capabailities are highly dependent on the application's programming model. Therefore, there has to be built-in library or language-level support for migration and malleability to allow applications to be reconfigurable. Part of this research consists of building the necessary tools to allow message passing applications to become reconfigurable with IOS. A library for process checkpointing, migration and malleability (PCM) has been designed and developed for MPI

iterative applications. The PCM is a user-level library that provides checkpointing, profiling, migration, split, and merge capabilities for a large class of iterative applications. Programmers need to specify the data structures that must be saved and restored to allow process migration and to instrument their application with few PCM calls. PCM also provides process split and merge functionalities to MPI programs. Common data distributions are supported like block, cyclic, and block-cyclic. PCM implements IOS generic profiling and reconfiguration interfaces, and therefore enables MPI applications to benefit from IOS reconfiguration policies.

The PCM API is simple to use and hides many of the intrinsic details of how to perform reconfiguration through migration, split and merge. Hence, with minimal code modification, a PCM-instrumented MPI application becomes malleable and ready to be reconfigured transparently by the IOS middleware. In addition, legacy MPI applications can benefit tremendously from the reconfiguration features of IOS by simply inserting a few calls to the PCM API.

1.3 Thesis Contributions

This research has generated a number of original contributions. They are summarized as follows:

1. A modular middleware for application reconfiguration with the goal of maintaining a reasonable performance in a dynamic environment. The modularity of our middleware is demonstrated through the use of several reconfiguration and coordination algorithms.

2. Fine-grained reconfiguration that enables reasoning about application entities rather than the entire application and therefore provides more concurrent and efficient adaptation of the application.

3. Decentralized and scalable coordination strategies for middleware agents that are based on partial knowledge.

4. Generic reconfiguration interfaces for application-level profiling and reconfiguration decisions to allow increased and portable adoption by several programming models and languages. This has been demonstrated through the successful

reconfiguration of actor-oriented programs in SALSA and process-oriented programs using MPI.

5. A portable protocol for inter-operation and interaction between applications and the middleware to ease the transition to reconfigurable execution in grid environments.

6. A user-level checkpointing and migration library for MPI applications to help develop reconfigurable message passing applications.

7. The use of *split* and *merge* or *malleability* as another reconfiguration mechanism to complement and enhance application adaptation through migration. Support of malleability in MPI applications developed by this research is the first of its kind in terms of splitting and merging MPI processes.

8. A resource-sensitive model for deciding when to migrate, split, or merge application's entities. This model enables reasoning about computational resources in a unified manner.

1.4 Thesis Roadmap

The remainder of this thesis is organized as follows:

- **Chapter 2** discusses background and related work in the context of dynamic reconfiguration of applications in grid environments. It starts by giving a literature review of emerging grid middleware systems and how they address resource management. It then reviews existing efforts for application adaptation in dynamic grid environments. An overview of programming models that are suitable for grid environments is given. Finally background information about world-wide computing, the actor model of computation and the SALSA programming language is presented.

- **Chapter 3** starts first by presenting key design goals that have been fundamental to the implementation of the Internet Operating System (IOS) middleware. These include operational and architectural goals at both the middleware-level

and application-level. Then the architecture of IOS is explained. This chapter also explains in details the various modules of IOS and its generic interfaces for profiling and reconfiguration.

- **Chapter 4** presents the different protocols and policies implemented as part of the middleware infrastructure. The protocols deal with coordinating the activities of middleware agents and forwarding work-stealing requests in a peer-to-peer fashion. At the core of the adaptation strategies, a resource-sensitive model is used to decide when, what and where to reconfigure application entities.

- **Chapter 5** explains how iterative MPI-based applications are reconfigured using IOS. First the checkpointing, migration, and malleability (PCM) library is presented. The chapter then proceeds by showing how the PCM library is integrated with IOS and the protocol used to achieve this integration.

- **Chapter 6** presents various kinds of experiments conducted in this research and the results obtained. In the first section, the performance evaluation of the middleware is given including evaluation of the protocols, scalability, and overhead. The second section presents various experiments that evaluate the reconfiguration functionalities of IOS using migration, split and merge, and a combination of them.

- **Chapter 7** concludes this thesis with a discussion of future directions.

CHAPTER 2

Background and Related Work

The deployment of grid infrastructures is a challenging task that goes beyond the capabilities of application developers. Specialized grid middleware is needed to mitigate the complex issues of integrating a large number of dynamic, heterogeneous, and widely distributed resources. Institutions need sophisticated mechanisms to leverage and share their existing information infrastructures in order to be part of public collaborations.

Grid middleware should address the following issues:

- **Security**. The absence of central management and the open nature of grid resources result in having to deal with several administrative domains. Each one of them brings along different resource access policies and security requirements. Being able to access externally-owned resources requires having the necessary credentials required by the external organizations. Users should be able to log on once and execute applications across various domains [1]. Furthermore, common methods for negotiating authorization and authentication are also needed.

- **Resource management**. Resource management is a fundamental issue for enabling grid applications. It deals with job submission, scheduling, resource allocation, resource monitoring, and load balancing. Resources in the grid have a very transient nature. They can experience constantly changing loads and avail-

[1] This capability of being able to log on once instead of logging in all used machines is referred to in the computational grid community as the *single sign-on policy*.

ability because of shared access and the absence of tight user control. Reliable and efficient execution of applications on such platforms requires application adaptation to the dynamic nature of the underlying grid environments. Adaptive scheduling and load balancing are necessary to achieve high performance of grid applications and high utilization of systems' resources.

- **Data management**. Since grid infrastructures involve widely distributed resources, data and processing might not necessarily be collocated. Concerns in data management arise such as how to efficiently distribute, replicate, and access potentially massive amounts of data.

- **Information management**. Being able to make informed decisions by resource managers requires the ability to discover available resources and learn about their characteristics (capacities, availability, current utilization, access policies, etc). Grid information services should allow the monitoring and discovery of resources and should make them available when necessary to grid resource managers.

This chapter focuses mainly on existing research in the area of grid resource management. The chapter is organized as follows. Section 2.1 surveys existing grid middleware systems. Section 2.2 discusses related work in resource management in grid systems. Section 2.3 discusses existing work in adaptive execution of grid applications. Section 2.4 presents various programming models that are good candidates for developing grid applications. In Section 2.5, we review basic peer-to-peer concepts and how they have been used in grid systems. Finally, Section 2.6 gives an overview about the worldwide computing project and presents several key concepts that have been fundamental to this dissertation.

2.1 Grid Middleware Systems

Over the last few years, several efforts have focused on building the basic software tools to enable resource sharing within scientific collaborations. Among these efforts, the most successful have been Globus [45], Condor [105], and Legion [29].

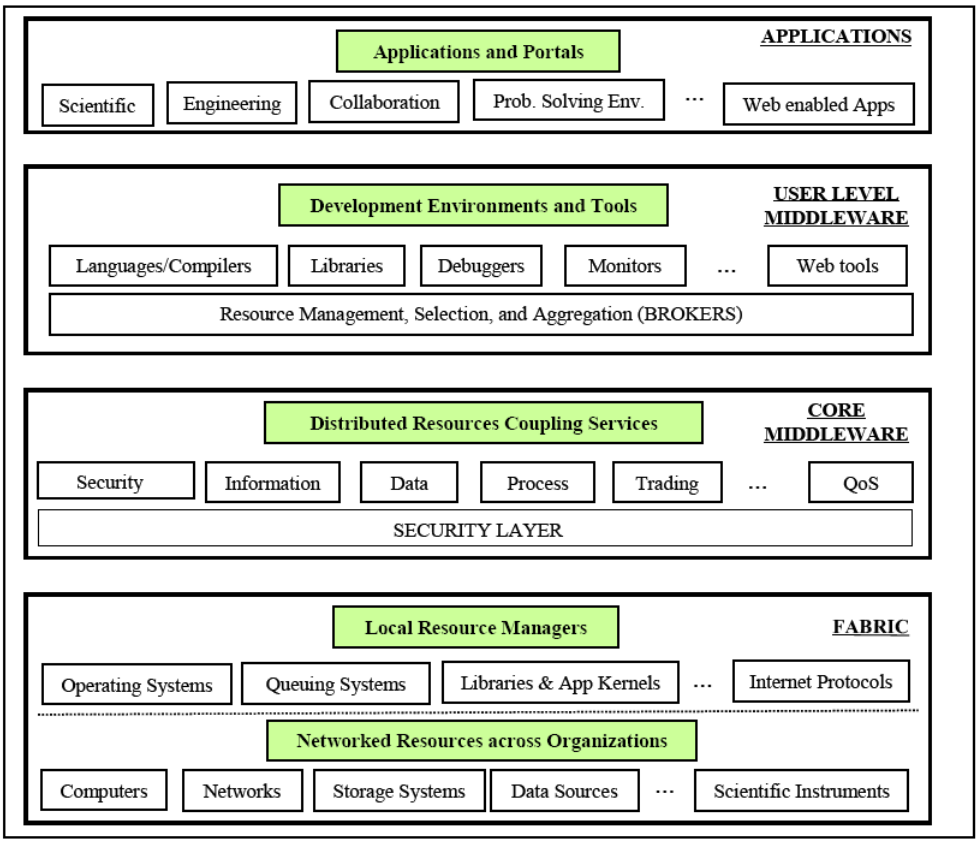

Figure 2.1: A layered grid architecture and components (Adapted from [14]).

The Globus toolkit has emerged as the *de-facto* standard middleware infrastructure for grid computing. Globus defines several protocols, APIs, and services that provide solutions to common grid deployment problems such as authentication, remote job submission, resource discovery, resource access, and transfer of data and executables. Globus adopts a layered service model that is analogous to the layered network model. Figure 2.1 shows the layered grid architecture that the Globus project adopts. The different layers of this architecture are briefly described in Table 2.1.

Condor is a distributed resource management system that is designed to support high-throughput computing by harvesting idle resource cycles. Condor discovers idle resources in a network and allocates them to application tasks [104]. Fault-tolerance is also supported through checkpointing mechanisms.

Legion specifies an object-based virtual machine environment that transparently

Layer	Description
Grid Fabric	Distributed resources such as clusters, machines, supercomputers, storage devices, scientific instruments, etc.
Core Middleware	A bag of services that offer remote process management, allocation of resources from different sites to be used by the same application, storage management, information registration and discovery, security, and Quality of Service (QoS).
User Level Middleware	A set of interfaces to core middleware services to provide higher levels of abstractions to end applications. These include resource brokers, programming tools, and development environments.
Grid Applications	Applications developed using grid-enabled programming models such as MPI.

Table 2.1: Layers of the grid architecture.

integrates grid system components into a single address space and file system. Legion plays the role of a grid operating system by addressing issues such as process management, input-output operations, inter-process communications, and security [80].

Condor-G [49] combines both Condor and Globus technologies. This merging combines Condor's mechanisms for intra-domain resource management and fault-tolerance with Globus protocols for security, and inter-domain resource discovery, access, and management. Entropia [30] is another popular system that utilizes cycle harvesting mechanisms. Entropia adopts similar mechanisms to Condor for resource allocation, scheduling and job migration. However, it is tailored only for Microsoft Windows 2000 machines, while Condor is tailored for both Unix and Windows platforms. WebOS [111] is another research effort with the goal of providing operating system's services to wide area applications, such as, resource discovery, remote process execution, resource management, authentication, security, and a global namespace.

2.2 Resource Management in Grid Systems

The nature of grid systems has dictated the need to come up with new models and protocols for grid resource management. Grid resource management differs

from conventional cluster resource management in several aspects. In contrast to cluster systems, grid systems are inherently more complex, dynamic, heterogeneous, autonomous, unreliable, and scalable. Several requirements need to be met to achieve efficient resource management in grid systems:

- **Site autonomy**. Traditional resource management systems assume tight control over the resources. These assumptions make it easier to design efficient policies for scheduling and load balancing. Such assumptions disappear in grid systems where resources are dispersed across several administrative domains with different scheduling policies, security mechanisms, and usage patterns. Additionally, resources in grid systems have a non-deterministic nature. They might join or leave at any time. It is therefore critical for a grid resource management system to take all these issues into account and preserve the autonomy of each participating site.

- **Interoperability**. Several sites use different local resource management systems such as the Portable Batch System (PBS), Load Sharing Facility (LSF), Condor, etc. Meta-schedulers need to be built that are able to interface and inter-operate with all the different local resource managers.

- **Flexibility and extensibility**. As systems evolve, new policies get implemented and adopted. The resource management system should be extensible and flexible to accommodate newer systems.

- **Support of negotiation**. QoS is an important requirement to meet several application requirements and guarantees. Negotiation between the different participating sites is needed to ensure that the local policies will not be broken and that the running applications will satisfy their requirements with certain guarantees.

- **Fault tolerance**. As systems grow in size, the chance of failures becomes non-negligible. Replication, checkpointing, job restart, or other forms of fault-tolerance have become a necessity in grid environments.

System	Architecture	Resource Discovery/ Dissemination/	QoS and Profiling	Scheduling
Globus	Hierarchical	Decentralized query discovery, periodic push dissemination	Partial support of QoS, state estimation relies on external tools such as NWS for profiling and forecasting of resources' performance	Decentralized, uses external schedulers for intra-domain scheduling
Condor	Flat	Centralized query discovery, periodic push dissemination	No support of QoS, matchmaking between client requests and resources' capabilities	Centralized
Legion	Hierarchical	Decentralized query discovery, periodic pull dissemination	Partial support of QoS, several schedules could be generated, the best is selected by the scheduler	Hierarchical

Table 2.2: Characteristics of some grid resource management systems.

- **Scalability**. All resource management algorithms should avoid centralized protocols to achieve more scalability. Peer-to-peer and hierarchical approaches will be good candidate protocols.

The subsequent section surveys main grid resource management systems, how they have addressed some of the above discussed issues, and their limitations. For each system, we will discuss its mechanisms for resource dissemination, discovery, scheduling, and profiling. Resource dissemination protocols can be classified as either using push or pull models. In the push model, information about resources is periodically pushed to a database. The opposite is done in a pull model, where resources are periodically probed to collect their information about them. Table 2.2 provides a summary of some characteristics of the resource management features of the surveyed systems.

2.2.1 Resource Management in Globus

A Globus resource management system consists of resource brokers, resource *co-allocators*, and resource managers, also referred to as Globus Resource Alloca-

tion Managers (GRAM). The task of *co-allocation* refers to allocating resources from different sites or administrative domains to be used by the same application. Dissemination of resources is done through an information service called the Grid Information Service (GIS), also known as the Monitoring and Discovery Service (MDS) [36]. MDS uses the Lightweight Directory Access Protocol [34] (LDAP) to interface with the gathered information about resources. MDS stores information about resources such as the number of CPUs, the operating systems used, the CPU speeds, the network latencies, etc. MDS consists of a Grid Index Information Service (GIIS) and a Grid Resource Information Service (GRIS). GRIS provides resource discovery services such as gathering, generating, and publishing data about resource characteristics in an MDS directory. GIIS tries to provide a global view about the different information gathered from various GRIS services. The aim is to make it easy for grid applications to look-up desired resources and match them to their requirements. GIIS indexes the resources in a hierarchical name space organization. Resource information is updated in GIIS by push strategies. Resource brokers discover resources by querying MDS.

Globus relies on local schedulers that implement Globus interfaces to resource brokers. These schedulers could be application-level schedulers (e.g. AppleS [16]), batch systems (e.g. PBS), etc. Local schedulers translate application requirements into a common language, called Resource Specification Language (RSL). RSL is a set of expressions that specify the jobs and the characteristics of the resources required to run them. Resource brokers are responsible for taking high level descriptions of resource requests and translating them into more specialized and concrete specifications. The transformed request should contain concrete resources and their actual locations. This process is referred to as *specialization*.

Specialized resource requests are passed to co-allocators who are responsible for allocating requests at multiple sites to be used simultaneously by the same application. The actual scheduling and execution of submitted jobs is done by the local schedulers. GRAM authenticates the resource requests and schedules them using the local resource managers. GRAM tries to simplify the development and deployment of grid applications by providing common APIs that hide the details of local schedulers, queuing systems, interfaces, etc. Grid users and developers do not need to know all

the details of other systems. GRAM acts as an entry point to various implementations of local resource management. It uses the concept of the hour-glass where GRAM is the neck of the hourglass, with applications and higher-level services (such as resource brokers or meta-schedulers) above it and local control and access mechanisms below it.

To sum up, Globus provides a bag of services to simplify resource management at a meta-level. The actual scheduling needs still be done by the individual resource brokers. Ensuring that an application efficiently uses resources from various sites is still a complex task. Developers still need to bear the burden of understanding the requirements of the application, the characteristics of the grid resources, and the optimal ways of scheduling the application using dynamic grid resources to achieve high performance.

2.2.2 Resource Management in Condor

The philosophy of Condor [105] is to maximize the utilization of machines by harvesting idle cycles. A group of machines managed by a Condor scheduler is called a Condor *pool*. Condor uses a centralized scheduling management scheme. A machine in the pool is dedicated to scheduling and information management. Submitted jobs are queued and transparently scheduled to run on the available machines of the pool. Job resource requests are communicated to the manager using the Classified Advertisements (ClassAds) [35] resource specification language [2]. Attributes such as processor's type, operating system, and available memory and disk space are used to indicate jobs' resource requirements.

Resource dissemination is done through periodic push mechanisms. Machines periodically advertise their capabilities and their job preferences in advertisements that use also ClassAds specification language. When a job is submitted to the Condor scheduler by a client machine, matchmaking is used to find the jobs and machines that best suit each other. Information about the chosen resources is then returned to the client machine. A shadow process is forked in the the client machine to take care of transferring executables and I/O redirection.

[2] The resource specification language used by Globus follows Condor's model for ClassAds. However Globus's language is more flexible and expressive.

Flocking [39] is an enhancement of Condor to share idle cycles across several administrative domains. Flocking allows several pool managers to communicate with one another and to submit jobs across pools. To overcome the problems of not having shared file systems, a split-execution model is used: I/O commands generated by a job are captured and redirected to the shadow process running on the client machine. This technique avoids transferring files or mounting foreign file systems.

Condor supports job preemption and check-pointing to preserve machine autonomy. Jobs can be preempted and migrated to other machines if their current machines decide to withdraw from the computation.

Condor adopts the philosophy of high throughput computing (HTC) as opposed to high performance computing (HPC). In HTC systems, the objective is to maximize the throughput of the entire system as opposed to maximizing the individual application response time in HPC systems. A combination of both paradigms should exist in grids to achieve efficient execution of multi-scale applications. Improving utilization and overall response and running time for large multi-scale applications are both important to justify the applicability for grid environments. On the one hand, application users will not be willing to use grids unless they expect to improve dramatically their performance. On the other hand, the grid computing vision tries to minimize idle resources by allowing resource sharing at a large scale.

2.2.3 Resource Management in Legion

Legion [29] is an object-based system that provides an abstraction over wide area resources as a worldwide virtual computer by playing the role of a grid operating system. It provides some of the traditional features that an operating system provides in a grid setting such as a global namespace, a shared file system, security, process creation and management, I/O, resource management, and accounting [80]. Everything in Legion is an object, an active process that reacts to function invocations from other objects in the system. Legion provides high-level specifications and protocols for object interaction. The implementations still have to be done by the users. Legion objects are managed by their own class object instances. The class object is responsible for creating new instances, activating or deactivating them, and schedul-

ing them. Legion defines core objects that implement system-level mechanisms: *host objects* represent compute resources while *vault* objects represent persistent storage resources.

The resource management system in Legion consists of four components: a scheduler, a schedule implementor, a resource database, and the pool of resources. Resource dissemination is done through a push model. Resources interact with the resource database, also called the *collection*. Users or schedulers obtain information about resources by querying the collection. For scalability purposes, there might be more than one collection object. These objects are capable of exchanging information about resources. Scheduling in Legion has a hierarchical structure. Higher level schedulers schedule resources on clusters, while lower-level schedulers schedule jobs on the local resources. When a job is submitted, an appropriate scheduler (Application-specific scheduler or a default scheduler) is selected from the framework. The scheduler, also called the *enactor object* is responsible for enforcing the schedule generated. There might be more than one schedule generated. The best schedule is selected. When it fails the next best is tried until all the schedules are exhausted.

Similar to Globus, Legion provides a framework for creating, and managing processes in a grid setting. However, achieving high performance is still a job that needs to be done by the application developers to make efficient use of the overall framework.

2.2.4 Other Grid Resource Management Systems

Several resource management systems for grid environments exist beside the discussed systems. 2K [65], is a distributed operating system that is based on CORBA. It addresses the problems of resource management in heterogeneous networks, dynamic adaptability, and configuration of entity-based distributed applications [64]. Bond [60] is a Java distributed agents system. The European DataGrid [56] is a Globus-based system for the storage and management of data-intensive applications. Nimrod [5] provides a distributed computing system for parametric modeling that supports a large number of computational experiments. Nimrod-G [25] is an extension of Nimrod that uses the Globus services and that follows a computational

economical model for scheduling. NetSolve [27] is a system that has been designed to solve computational science problems through a client-agent-server architecture. WebOS seeks to provide operating system services, such as client authentication, naming, and persistent storage to wide area applications [111].

There is a large body of research into computational grids and grid-based middleware, hence this section only attempts to discuss a selection of this research area. The reader is referred to [66] and [115] for a more comprehensive survey of systems geared toward distributed computing on a large-scale.

2.3 Adaptive Execution in Grid Systems

Globus middleware provides services needed for secure multi-site execution of large-scale applications gluing different resource management systems and access policies. The dynamic and transient nature of grid systems necessitates adaptive models to enable the running application to adapt itself to rapidly changing resource conditions. Adaptivity in grid computing has been mainly addressed by adaptive application-level scheduling and dynamic load balancing. Several projects have developed application-oriented adaptive execution mechanisms over Globus to achieve an efficient exploitation of grid resources. Examples include AppleS [17], Cactus-G [12], GrADS [40], and GridWay [59]. These systems share many features with differences in the ways they are implemented.

AppleS [17] applications rely on structural performance models that allows prediction of application performance. The approach incorporates static and dynamic resource information, performance predictions, application and user-specific information and scheduling techniques to adapt to application's execution "on-the-fly". To make this approach more generic and reusable, a set of template-based software for a collection of structurally similar applications has been developed. After performing the resource selection, the scheduler determines a set of candidate schedules based on the performance model of the application. The best schedule is selected based on user's performance criteria such as execution time and turn-around time. The schedule generated can be adapted and refined to cope with the changing behavior of resources. Jacobi2D [18], Complib [94], and Mcell [28] are examples of applications

that benefited from the application-level adaptive scheduling of AppleS.

Adaptive grid execution has been also explored in the Cactus project through support of migration [69]. Cactus is an open source problem-solving environment designed for solving partial differential equations. Cactus incorporates, through special components referred to as *grid-aware thorns* [11], adaptive resource selection mechanisms to allow applications to change their resource allocations via migration. Cactus uses also the concept of contract violation. Application migration is triggered whenever a contract violation is detected and the resource selector has identified alternative resources. Checkpointing, staging of executables, allocation of resources, and application restart are then performed. Some application-specific techniques were used to adapt large applications to run on the grid such as adaptive compression, overlapping computation with communication, and redundant computation [12].

The GrADS project [40] has investigated also adaptive execution in the context of grid application development. The goal of the GrADS project is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing. Grid application development in GrADs involves the following components: 1) Resource selection is performed by accessing Globus MDS and the Network Weather Service [117] to get information about the available machines, 2) An application-specific performance modeler is used to determine a good initial matching list for the application, 3) and a contract monitor, which detects performance degradation and accordingly does rescheduling to re-map the application to better resources. The main components involved in application adaptation are the contract monitor, the migrator, and the rescheduler which decides when to migrate. The migrator component relies on application-support to enable migration. The Stop Restart Software (SRS) [110] is a user-level checkpointing library used to equip the application with the ability to be stopped, checkpointed, and restarted with a different configuration. The Rescheduler component allows migration on-request and opportunistic migration. Migration cost is evaluated by considering the predicted remaining execution time in the new configuration, the current remaining execution time, and the cost of rescheduling. Migration happens only if the gain is greater than a 30% threshold [109].

In the GridWay [59] project, the application specifies its resource requirements and ranks the needed resources in terms of their importance. A submission agent automates the entire process of submitting the application and monitoring its performance. Application performance is evaluated periodically by running a performance degradation evaluator program and by evaluating the accumulated suspension time. The application has a job template which contains all the necessary parameters for its execution. The framework evaluates if migration is worthwhile or not in case of a rescheduling event. The submission manager is responsible for the execution of a job during its lifetime. It is also responsible for performing job migration to a new resource. The framework is responsible for submitting jobs, preparing RSL files, performing resource selection, preparing the remote system, canceling the job in case of a kill, stop, or migration event. When performance slowdown is detected, rescheduling actions are initiated to detect better resources. The resource selector tries to find jobs that minimize total response time (file transfer and job execution). Application-level schedulers are used to promote the performance of each individual application without considering the rest of the applications. Migration and rescheduling could be user-initiated or application-initiated.

2.4 Grid Programming Models

To achieve an application adaptation to grid environment, not only should the middleware provide the necessary means for state estimation, and reconfiguration of resources. The application's programming model should also allow the application to react to the different reconfiguration requests from the underlying environment. This functionality can take different forms, such as application migration, process migration, checkpointing, replication, partitioning, or change of application granularity.

We describe in what follows existing programming models that appear to be relevant to grid environments and that provide some partial support for reconfiguration.

- **Remote procedure calls (RPC)** [20]. RPC uses the client-server model to implement concurrent applications that communicate synchronously. The RPC mechanism has been traditionally tailored for single-processor systems

and tightly coupled homogeneous systems. GridRPC is a collaboration effort to extend the RPC model to support grid computing and to standardize its interfaces and protocols. The extensions consist basically of providing support for coarse-grained asynchronous systems. NetSolve [27] is a current implementation of GridRPC [92] based on a client-agent-server system. The role of the agent is to locate suitable resources and select the best ones. Load balancing policies are used to attempt a fair allocation of resources. Ninf [91] is another implementation built on top of Globus services.

- **Java-based models.** Java is a powerful programming environment for developing platform-independent distributed systems. It was originally designed with distributed computing in mind. The applet and Remote Method Invocation (RMI) models are some features of this design. The use of Java in grid computing has gained even more interest since the introduction of web services in the OGSI model. Several APIs and interfaces are being developed and integrated with Java. The Java-Grande project is a huge collaborative effort trying to bring Java platform up-to-speed for high-performance applications. The Java Commodity Grid (CoG) toolkit [70] is another effort for providing Java-based services for grid-computing. CoG provides an object-oriented interface to standard Globus toolkit services.

- **Message passing**. This model is the most widely used programming model for parallel computing. It provides application developers with a set of primitive tools that allow communication between the different tasks, collective operations like broadcasts and reductions, and synchronization mechanisms. However, message passing is still a low-level paradigm and does not provide high-level abstractions for task parallelism. It requires a lot of expertise from developers to achieve high performance. Popular message passing libraries include MPI and the Parallel Virtual Machine (PVM) [41]. MPI has been implemented successfully on massively parallel processors (MPPS) and supports a wide range of platforms. However, existing portable implementations target homogeneous systems and have very limited support for heterogeneity. PVM provides sup-

port for dynamic addition of nodes and host failures. However, its limited ability to meet the required high performance on tightly coupled homogeneous systems did not encourage a wide adoption. Extensions to MPI to meet grid requirements have been actively pursued recently. MPICH-G2 is a grid-enabled implementation of MPI based on MPICH, a portable implementation of MPI. MPICH-G2 is built upon the Globus toolkit. MPICH-G2 allows the use of multiple heterogeneous machines to execute MPI applications. It automatically converts data in messages sent between machines of different architectures and supports multi-protocol process communication through automatic selection of TCP for inter-machine messaging and more highly optimized vendor-supplied MPI implementations (whenever available) for intra-machine messaging.

- **Actor model** [7, 54]. An actor is an autonomous entity that encapsulates state and processing. Actors are concurrent entities that communicate asynchronously. Processing in actors is solely triggered by message reception. In response to a message, an actor can change its current state, create a new actor, or send a message to other actors. The anatomy of actors facilitates autonomy, mobility, and asynchronous communication and makes this model attractive for open distributed systems. Several languages and frameworks have implemented the Actor model (e.g., SALSA [114], Actor Foundry [81], Actalk [23], THAL [63] and Broadway [97]). A more detailed discussion of the Actor model and the SALSA language is given in Section 2.6.

- **Parallel Programming Models.** Several models have emerged to abstract application parallelism on distributed resources. The Master-Worker (MW) model is a traditional parallel scheme whereby a master task defines dynamically the tasks that must be executed and the data on which they operate. Workers execute the assigned tasks and return the result to the master. This model exhibits a very large degree of parallelism because it generates a dynamic number of independent tasks. This model is very well suited for grids. The AppleS Master Worker Application Template (AMWT) provides adaptive scheduling policies for MW applications. The goal is to select the best

placement of the master and workers on grid resources to optimize the overall performance of the application. The Fork-join model is another model where the degree of parallelism is dynamically determined. In this model, tasks are dynamically spawned and data is dynamically agglomerated based on system characteristics such as the amount of workload or the availability of resources. This model employs a two-level scheduling mechanism. First a number of virtual processors are scheduled on a pool of physical processors. The virtual processors represent kernel-level threads. Then user-level threads are spawned to execute tasks from a shared queue. The forking and joining is done at the user-level space because it is much faster than the kernel-level thread. Several systems have implemented this model such as Cray Multitasking [88], Process Control [51], and Minor [76]. All the afore mentioned implementations have been targeted mainly for shared-memory and tightly coupled systems. Other effective parallel programming models have been studied, such as divide and conquer applications and branch and bound. The Satin [112] system is an example of a hierarchical implementation of the divide and conquer paradigm targeted for grid environments.

2.5 Peer-to-Peer Systems and the Emerging Grid

Grid and peer-to-peer systems share a common goal: sharing and harnessing resources across various administrative domains. However they both evolved from different communities and provide different services. Grid systems focus on providing a collaborative platform that interconnects clusters, supercomputers, storage systems, and scientific instruments from trusted communities to serve computationally intensive scientific applications. Grid systems are of moderate size and are centrally or hierarchically administered. Peer-to-peer (P2P) systems provide intermittent participation for significantly larger untrusted communities. The most common P2P applications are file sharing and search applications. It has been argued that grid and P2P systems will eventually converge [48, 99]. This convergence will likely happen when the participation in grid increases to the scale of P2P systems, when P2P systems will provide more sophisticated services, and when the stringent QoS require-

ments of grid systems are loosened as grids host more popular user applications. In what follows, we give an overview of P2P systems. Then we give an overview of some research efforts that have tried to utilize P2P techniques to serve grid computing.

The peer-to-peer paradigm is a successful model that has been proved to achieve scalability in large-scale distributed systems. As opposed to traditional client-server models, every component in a P2P system assumes the same responsibilities acting as both a client and a server. The P2P approach is intriguing because it has managed to circumvent many problems with the client/server model with very simple protocols. There are two categories of P2P systems based on the way peers are organized and on the protocol used: unstructured and structured. Unstructured systems impose no structure on the peers. Every peer in an unstructured system is randomly connected to a number of other peers (e.g. Examples include Napster [3], Gnutella [33], and KaZaA [2]). Structured P2P systems adopt a well-determined structure for interconnecting peers. Popular structured systems include Chord [79], Pastry [74], Tapestry [119], and CAN [98].

In a P2P system, peers can be organized in various topologies. These topologies can be classified into centralized, decentralized and hybrid. Several p2p applications have a centralized component. For instance, Napster, the first file sharing application that popularized the P2P model, has a centralized search architecture. However, the file sharing architecture is not centralized. The SETI@Home [13] project has a fully centralized architecture. SETI@Home is a project that harnesses free CPU cycles across the Internet (SETI is an acronym of Search for Extra-Terrestrial Intelligence). The purpose of the project is to analyze radio telescope data for signals from extra-terrestrial intelligence. One advantage of the centralized topology is the high performance of the search because all the needed information is stored in one central location. However, this centralized architecture creates a bottleneck and cannot scale to a large number of peers. In the decentralized topology, peers have equal responsibilities. Gnutella [33] is among the few pure decentralized systems. It has only an initial centralized bootstrapping mechanism by which new peers learn about existing peers and join the system. However, the search protocol in Gnutella is completely decentralized. Freenet is another application with a pure decentralized topology. With

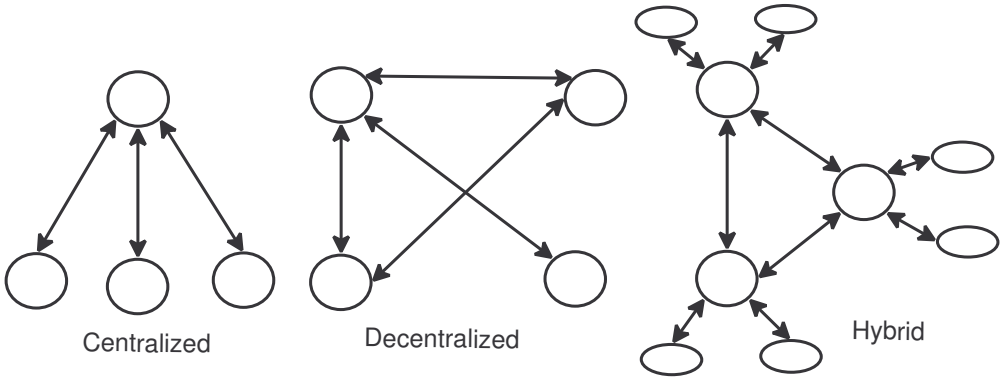

Figure 2.2: Sample peer-to-peer topologies: centralized, decentralized and hybrid topologies.

decentralization comes the cost of having a more complex and more expensive search mechanism. Hybrid approaches emerged with the goal of addressing the weaknesses of centralized and decentralized topologies, while benefiting from their advantages. In a hybrid topology, peers have various responsibilities depending on how important they are in the search process. An example of a hybrid system is the KazaA [2] system. KazaA is a hybrid of the centralized Napster and the decentralized Gnutella. It introduced a very powerful concept: *super peers*. Super peers act as local search hubs. Each super peer is responsible for a small portion of the network. It acts as a Napster server. These special peers are automatically chosen by the system depending on their performance (storage capacity, CPU speed, network bandwidth, etc) and their availability. Figure 2.2 shows example centralized, decentralized, and hybrid topologies.

P2P approaches have been mainly used to address resource discovery and presence management in grid systems. Most current resource discovery mechanisms are based on hierarchical or centralized schemes. They also do not address large scale dynamic environments where nodes join and leave at anytime. Existing research efforts have borrowed several P2P dynamic peer management and decentralized protocols to provide more dynamic and scalable resource discovery techniques in grid systems. In [48] and [100], a flat P2P overlay is used to organize the peers. Every virtual organization (VO) has one or more peers. Peers provide information about one or

more resources. In [48], different strategies are used to forward queries about resource characteristics such as random walk, learning-based, and best-neighbor. A modified version of the Gnutella protocol is used in [100] to route query messages across the overlay of peers. Other projects [75, 83] have adopted the notion of super peers to organize, in a hierarchical manner, information about grid resources. Structured P2P concepts have also been adopted for resource discovery in grids. An example is the MAAN [26] project that proposes an extension of the Chord protocol to handle complex search queries.

2.6 Worldwide Computing

Varela, et al. [10] introduced the notion and vision of the World-Wide Computer (WWC) which aims at turning the widespread resources in the Web into a virtual mega-computer with a unified, dependable and distributed infrastructure. The WWC provides naming, mobility, and coordination constructs to facilitate building widely distributed computing systems over the Internet. The architecture of the WWC consists of three software components: 1) SALSA, a programming language for application development, 2) a distributed runtime environment with support for naming and message sending, and 3) a middleware layer with a set for services for reconfiguration, resource monitoring, and load balancing. Figure 2.3 shows the layered architecture of the World-Wide Computer.

The WWC views all software components as a collection of *actors*. The Actor model has been fundamental to the design and implementation of the WWC architecture. We discuss in what follows concepts related to the Actor model of computation and the SALSA programming language.

2.6.1 The Actor Model

The concept of *actors* was first introduced by Carl Hewitt at MIT in the 1970's to refer to autonomous reasoning agents [54]. The concept evolved further with the work of Agha and others to refer to a formal model of concurrent computation [9]. This model contrasts with (and complements) the object-oriented model by emphasizing concurrency and communication between the different components.

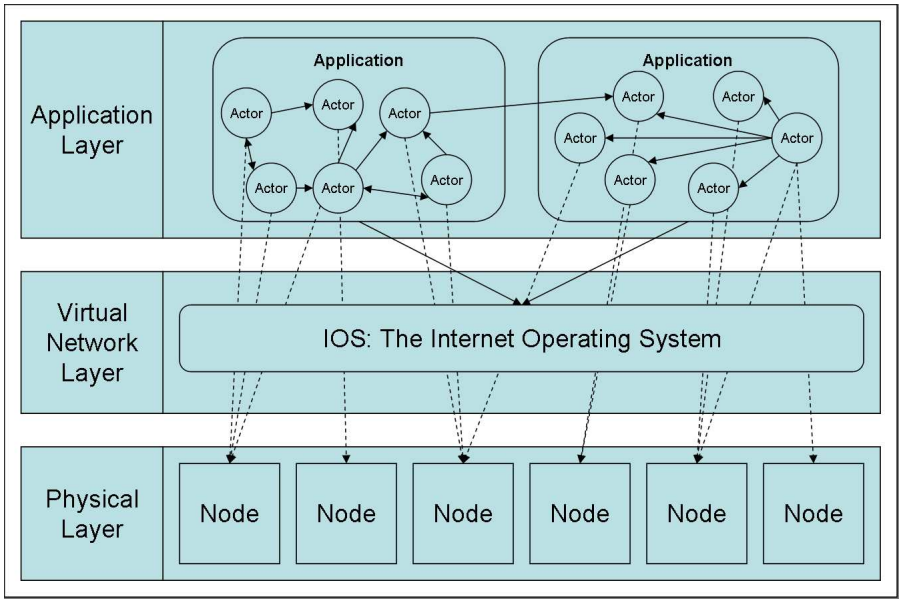

Figure 2.3: A Model for Worldwide Computing. Applications run on a virtual network (a middleware infrastructure) which maps actors to locations in the physical layer (the hardware infrastructure).

Actors are inherently concurrent and distributed objects that communicate with each other via asynchronous message passing. An actor is an object because it encapsulates a state and a set of methods. It is autonomous because it is controlled by a single thread of execution. When an actor receives a message, one of three primitive operations might happen (see Figure 2.4):

- changing its internal state through invoking one of its internal methods,
- sending a message to another actor, or
- creating a new actor.

Actor methods are atomic operations. In other words, while an actor is executing one of its methods, it cannot be interrupted by another message. An actor maintains a mailbox where it buffers received messages until it is ready to respond to them. There is no guarantee that messages will be processed in the order in which they were sent. The only guarantee is that any message will eventually be processed by the receiving actor.

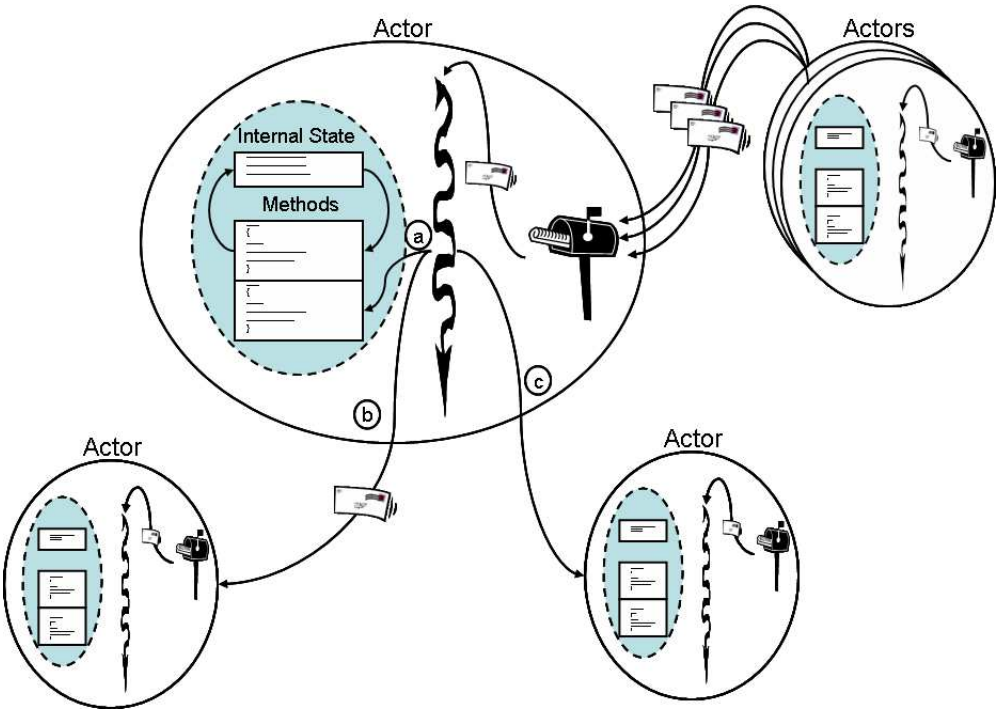

Figure 2.4: The primitive operations of an actor. In response to a message, an actor can: a) change its internal state by invoking one of its internal methods, b) send a message to another actor, or c) create a new actor.

The anatomy of actors simplifies concurrency, synchronization, coordination, namespace management, memory management, scheduling, and mobility in distributed systems. The actor model has been successfully used in several applications. To name a few: enterprise integration [106], real-time programming [87], fault-tolerance [8], and distributed artificial intelligence [43].

2.6.2 The SALSA Programming Language

SALSA [114] is a language for developing actor-oriented applications. SALSA is an acronym for **S**imple **A**ctor **L**anguage **S**ystem and **A**rchitecture. SALSA is preprocessed to Java. Therefore it inherits all the object-oriented features of Java such as inheritance, encapsulation, and polymorphism. SALSA provides programming abstractions to implement actor primitives notably creation and asynchronous communication. It has been designed with additional extensions to facilitate programming

```
module example.salsa.behavior;

behavior SalsaBehavior {

    //constructor
    void SalsaBehavior(){
       //do something
    }

    void message1(){
        //do something
    }
    void message2(){
        //do something
    }

    void act(String arguments[]){

        //Actor creation
        // Creates an actor with the SalsaBehavior behavior
        //with a given name at a specific location.

        SalsaBehavior myactor = new SalsaBehavior()
                                at (myUAN, myUAL);

        //Message sending
        myactor<-message1();

        //Send the message println to the actor sandardOutput
        standardOutput<-println("sample message");

        //Actor migration
        // This results into migrating the actor myactor to
        // the location newUAL
        myactor<-migrate(newUAL);

        //Coordination example
        //Enforcing the oder of message processing
        //the actor myactor should process message1 before
        //processing message 2
        myactor<-message1()@myactor<-message2();
    }
```

Figure 2.5: Skeleton of a SALSA program. The skeleton shows simple examples of actor creation, message sending, coordination, and migration.

Type	Example
URL	http://wcl.cs.rpi.edu/
UAN	uan://europa.wcl.cs.rpi.edu:3030/myID
UAL	rmsp://io.wcl.cs.rpi.edu:4040/myLocation

Table 2.3: Examples of a Universal Actor Name (UAN) and a Universal Actor Locator (UAL).

JAVA Concepts	Analogous SALSA Concepts
Object	Actor
Class	Behavior
Method	Message Handler
main Method	act Message
Method Invocation	Message Sending
Package	Module

Table 2.4: Some Java concepts and their analogous SALSA concepts.

reconfigurable distributed applications:

- **Universal naming**. Every actor has a unique name that is used as a point of contact by other actors. Message sending can simply be done by just knowing the universal name of an actor. In SALSA's terms, a *Universal Actor Name* (UAN) refers to the actor's unique name, while a *Universal Actor Locator* refers to its current location. The UAN and UAL formats follow the Uniform Resource Identifier (URI) syntax. Hence, they are very similar to a Uniform Resource Locator (URL) used by the WWW as the examples in Table 2.3 illustrate.

- **Migration**. SALSA's runtime environment provides the necessary support to perform migration of an actor. This feature provides actors with mobility; a necessary ingredient for reconfigurability in distributed systems.

- **Coordination primitives**. The language supports several mechanisms that coordinate the behavior of actors. More specifically, SALSA provides primitives for ordering messages, and delegating computations to third-parties.

Figure 2.5 shows a skeleton of a SALSA program with simple examples for actor creation, migration, message sending, and coordination. The equivalent of Java's

packages, objects, classes, and methods in SALSA are *modules*, *behaviors*, *actors*, and *messages* (see Table 2.4).

2.6.3 Theaters and Run-Time Components

Actors in the WWC framework rely on the Universal Actor Naming Protocol (UANP) and Remote Message Sending Protocol (RMSP) for communication. RMSP enables message sending and actor migration. UANP enables communication with naming authorities that map actor locations to human readable names. The UANP and RMSP protocols provide transport mechanisms for actor in a portable manner in similar fashion to the HTTP transport protocol in the World-Wide Web.

Theaters are nodes of the WWC that provide execution environments for actors. Each theater consists of an RMSP server, a mapping between relative actor locations and executing actor references, and a runtime environment. The theater might also contain stationary environmental actors that provide access to stationary resources such as standard output.

The WWC provides a naming service which enables messages to be sent to an actor as its location changes from one theater to another. The universal actor naming protocol consists of a small number of message types and is very extensible.

CHAPTER 3

A Middleware Framework for Adaptive Distributed Computing

We discuss in this chapter a modular framework for achieving adaptation based on dynamically reconfigurable *grid-aware* applications. An application is said to be grid-aware if it can sense the characteristics of the underlying grid environment and dynamically reconfigures its resources allocation or its structure to maintain a desired level of performance [3]. Dynamic reconfiguration implies the ability of the application to dynamically change the mappings of its entities to physical resources or the degree of parallelism it exposes in response to grid environment changes. The framework does not necessarily assume an initial knowledge of the performance model of the application. The latter can be learned through automatic profiling strategies.

The chapter starts by discussing in Section 3.1 key design issues at both the middleware-level and the application-level that need to be addressed to achieve successful reconfiguration for applications in dynamic environments. Section 3.2 describes a model for grid-aware reconfigurable applications. The architecture of the Internet Operating System middleware is explained in Section 3.3. Then a description of the profiling and reconfiguration interfaces of IOS are presented in Section 3.4.

[3]The term grid-aware was introduced first in the GrADS project [40].

3.1 Design Goals

Managing reconfiguration issues over large-scale dynamic environments is beyond the scope of application developers. It is therefore imperative to adopt a middleware-driven approach to achieve efficient deployment of distributed applications in a dynamic setting in a portable and transparent manner. One way of achieving this vision is embodying various middleware strategies and services within a virtual network of autonomous agents that coordinate applications and resources.

Several design issues need to be addressed to achieve middleware-trigged application reconfiguration. We discuss these goals at both the middleware-level and the application-level.

3.1.1 Middleware-level Issues

Architectural Modularity Middleware agents should provide mechanisms that allow profiling meta-level information about applications' behavior and resource usage characteristics, propagating this meta-level information, and deciding when and how to reconfigure applications. It is hard, if not impossible, to adopt one single strategy that is applicable to different types of execution environments and a large spectrum of applications. For example, it should be possible for the middleware to enable tuning the amount of profiling done depending on the load of the underlying resources. The more accurate the profiling is, the more informed the reconfiguration decisions are. A modular architecture is therefore critical to have extensible and pluggable reconfiguration mechanisms. Different profiling, coordination, and reconfiguration decisions strategies can be used depending on the class of applications and the characteristics of the underlying execution environment.

Generic Interfaces To allow middleware to manage applications written using various programming models and technologies, common and generic interfaces need to be designed. The functional details of how to accomplish reconfigurability such as migration, replication, or re-distribution of computation are different among various programming models. However, the strategies embodied in the middleware to

achieve application's adaptation such as deciding how to disseminate resource characteristics and how to use meta-level information about applications and resources to effectively reconfigure applications are similar and therefore reusable across various programming paradigms. Generalizing and standardizing the interactions between applications and middleware lead to a more generic and portable approach capable of using new programming models and easily upgrading legacy applications that use existing models to enable dynamic reconfiguration.

Decentralized and Scalable Strategies Grids consist of thousands—potentially millions—of globally distributed computational nodes. To address the demands of such large-scale environments with higher degrees of node failures and latency variability, decentralized management becomes a necessity. The sheer size of these computational environments renders the use of decisions based on global knowledge infeasible and impractical since the overhead of obtaining global information may overwhelm the benefits of using it. This forces any middleware to rely on making decisions based on local, partial, and often inaccurate information. There is an inherent trade-off between the overhead of information dissemination and the maintenance of high quality of information.

Dynamic Resource Availability Large-scale execution environments are expected to be more dynamic. Resource availability is constantly changing. Resources can join anytime during the lifetime of an application, for example, when users voluntarily engage their machines as part of a distributed computation over the Web, or when an over-loaded machine becomes suddenly available upon the completion of some tasks. Resources can also leave at anytime, for example, when a user withdraws her/his machine from a web-based computation or because of a node failure. To improve efficiency, middleware-driven reconfiguration can help applications adjust their resource allocation as the availability of the underlying physical resources changes.

3.1.2 Application-level Issues

The middleware's ability to reconfigure an application dynamically is ultimately bound by the application's reconfiguration capabilities. Different levels of reconfigura-

bility include application stop and restart mechanisms, application entity migration and replication, and dynamic granularity of application entities. The latter techniques require more complex support from application programming models and tend to be less transparent to application developers. However, they provide the most flexibility to middleware and the best opportunities for adaptability and scalability.

Functional issues of reconfiguration such as how to migrate application entities or how to change the granularity of entities are highly dependent on the programming model and language used to develop a given application. It is imperative to have tools that augment applications with these capabilities. When such tools are available, the middleware can readily use them to automate the process of application adaptation in dynamic environments. The middleware can shield application developers from dealing with complex reconfiguration issues such as when and where to migrate application's entities and which entities need to be reconfigured.

In programming models such as actors [7, 54], application entity migration is obtained virtually for free since memory is completely encapsulated and distributed and communication is performed purely by asynchronous message passing. In programming models such as message passing (e.g., MPI [77]), additional libraries and application programming interfaces (API) are required to enable process migration. In both the actor and process programming models, dynamic entity granularity requires help from the application developers since data repartitioning and process interconnectivity changes are usually highly application-dependent. Nonetheless, well-designed APIs can make the process of extending applications to support dynamic reconfiguration more manageable. This is particularly true for massively parallel applications composed of relatively independent subtasks.

3.2 A Model for Reconfigurable Grid-Aware Applications

3.2.1 Characteristics of Grid Environments

Reconfigurable grid-aware applications should be able to adapt themselves dynamically to the following grid characteristics:

- **Heterogeneous resources**. Resources have highly heterogeneous configurations: different architectural platforms, different CPU speeds, memory hierarchy

layouts and capacities, disk space, and software configurations.

- **Dynamic resource availability**. The absence of tight control on volunteer computing-based grid environments can cause a resource to withdraw from the computation at any time based on the will of the owner. Resources might non-deterministically join or leave the available set of resources. This renders the grid a best-effort computation medium by analogy to the Internet's best-effort communication medium.

- **Dynamic resource load**. Applications run in a shared environment, where external jobs compete for the same resources. Therefore the fluctuations of the machine loads are expected to be high. A machine might be lightly loaded now and heavily loaded in the next minute.

- **Increased failure rates**. The large number of resources makes the probability of hardware or software failure very high.

Grid applications need to have a flexible structure that synergizes easily with the dynamic nature of grids. They should exhibit a large degree of processing and/or data parallelism that allows for efficient usage of the grid and for scalability to a large number of resources. Two important issues need to be addressed to design and develop reconfigurable grid-aware applications: programming abstractions and efficient middleware mechanisms for reconfiguration. Programming models provide high-level abstractions that can be used to structure the application as distributed or parallel entities that cooperate to perform specific tasks. Middleware mechanisms encapsulate all the intrinsic details of run-time resource management and reconfiguration policies. The programming model should cooperate with the middleware environment and should enable the application to respond properly to middleware-triggered reconfiguration decisions. In Section 2.4, we discussed several programming models that meet some requirements of grid applications. Figure 3.1 shows the flow of information between the reconfigurable application and the middleware.

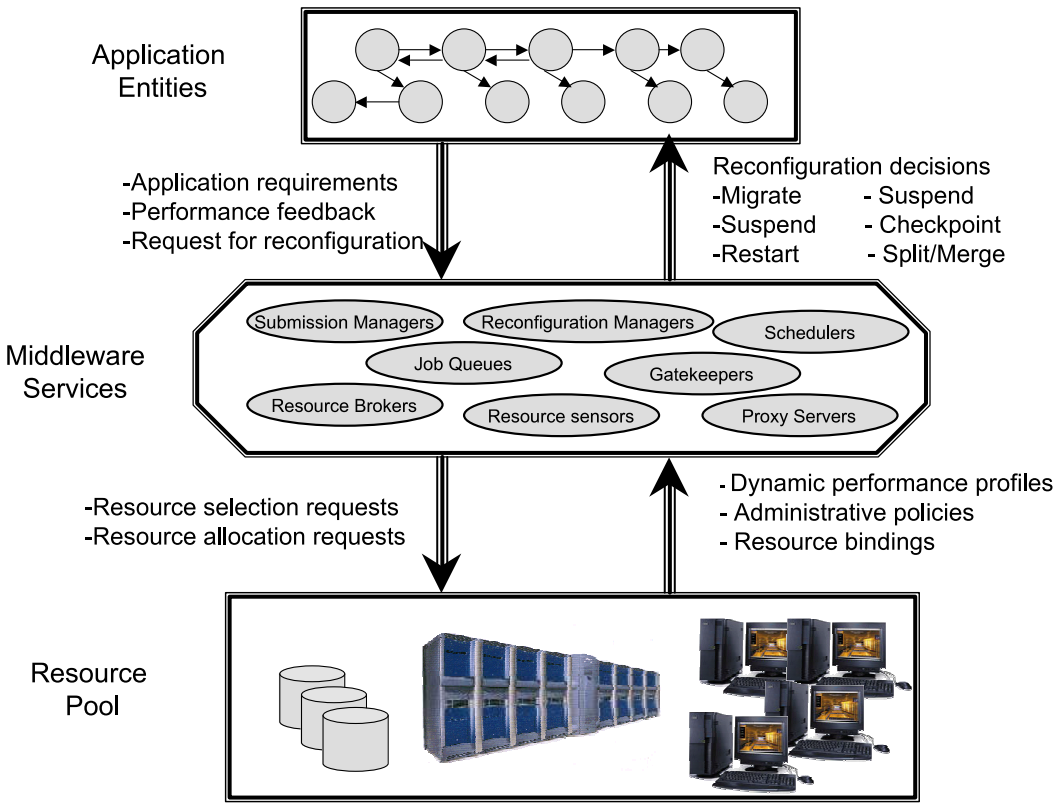

Figure 3.1: Interactions between reconfigurable applications, the middleware services, and the grid resources.

3.2.2 A Grid Application Model

A grid-aware reconfigurable application should consist of a set of autonomous entities that encapsulate enough information about their internal state and their connectivity with the rest of the entities to allow the middleware to reason about their state. Such information may include the current resources the entity is currently bound to, the performance requirements, the performance profile over a period of time, the predicted performance for a given period in the future, and its connectivity graph with the rest of the application.

In this section, we describe in formal terms the elements that constitute a reconfigurable application. The aim of the model is to provide high-level abstractions of the structure and behavior of the application to ease reconfiguration in a dynamic grid.

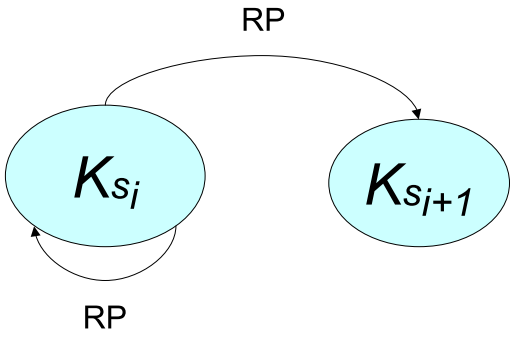

Figure 3.2: A state machine showing the configuration states of an application at reconfiguration points.

Definitions and Notations. The lifetime of the application may consist of a sequence of one or more execution stages, s_j. During each stage, the number of application entities, their mappings to physical resources, and their data distributions are constant. The transition from a stage s_j to another stage s_{j+1} happens only in the presence of a reconfiguration event. One or a combination of the following transformations may happen:

- A change in the mapping from the application entities to the physical resources through migration.
- A change in the number of entities: spawning, garbage-collecting, splitting, or merging entities.
- A change in the data distribution among the entities.

We call the state of an application during an execution state s, the application's *configuration*, denoted by K_s. We refer to the boundaries between the stages as *Reconfiguration Points*, RP's. The state of the application can be changed only at an RP. The result of an RP may lead to the same or a different configuration (see Figure 3.2).

In more formal terms, the application model consists of the following elements.

- A. The application to be reconfigured.

- S. The sequence of execution states.

$$S = \{s_0, \ldots, s_f\}$$

- C_s. The group of n concurrent entities c_i's of application A at stage s.

$$C_s = \{c_0, ..., c_{n-1}\}$$

- R_s. The current pool of m resources r_j's used by the application at stage s.

$$R_s = \{r_0, ..., r_{m-1}\}$$

- B_s. The bindings between the physical resources r_j's and the entities c_i's during stage s.

$$B_s = \{(c_i, r_j),\ 0 \leq i \leq n-1\ \wedge\ 0 \leq j \leq m-1\}$$

- $G_{A,s}$. The communication graph of the application A at a given stage s, $G_{A,s} = (C, E_A)$. $G_{A,s}$ is a directed and weighted graph, where the nodes represent the entities c_i's and the edges represent the communication links between the entities. An edge $e(c_i \rightarrow c_j)$ is associated with a weight $w_{i,j}$ that measures the communication rates between the the designated entities (e.g. bytes/s).

An entity maintains both static and dynamic information regarding its performance. A *performance profile* denotes dynamic data that records the performance progress of an entity during an execution stage. Performance profiles store information about the processing and communication progress of the entity, i.e, the processing and data rate experienced by the entity. A *performance specification* refers to the performance requirements of the entities, such as, i) how much data needs to be sent and received to make a unit of progress, ii) memory requirements, iii) CPU speed requirements, and iv) storage requirements. We assume for the time being that performance specifications are static data that do not change over the life-time of the entity. However, this assumption will no longer be valid if the application is evolutionary and has dynamic requirements. Formally, we model performance profiles and

performance specifications as follows:

- $G_{c,s}^{profile}$. The communication graph of entity c at a given stage s. This graph models the dynamic communication performance of entity c. It is a weighted graph of nodes, where the nodes represent the entities c_i that communicate with c during s and the edges represent the communication links between c and the other entities. Every edge $e(c \rightarrow c_j)$ is associated with a weight w_j that measures the average sending and receiving communication rates with entity c_j during s (e.g. bytes/s).

- $P_{c,s}^{profile}$. The performance profile of an entity c at stage s. Let $G_{c,s}^{profile}$ be the communication graph of entity c, $p_{c,s}$ be the average processing rate of c, $m_{c,s}$ be the average used memory, and $d_{c,s}$ be the average used disk storage. All measurements are done during stage s.

$$P_{c,s}^{profile} =< G_{c,s}^{profile}, m_{c,s}, p_{c,s}, d_{c,s} >$$

- G_c^{spec}. The communication specification graph of an entity c models its communication requirements. G_c is a weighted graph of nodes, where the nodes represent the entities c_i's that communicate with c and the edges represent the communication demands between c and the other entities. Every edge $e(c \rightarrow c_j)$ is associated with a weight w_j that specifies how much data needs to be sent or received for c to be able to make a unit of progress.

- P_c^{spec}. The performance specification of an entity c. Let G_c^{spec} be the communication specification graph of c, P_c be the minimum processing requirement of c, M_c be the minimum memory requirement of c, and D_c be the minimum disk requirement of c.

$$P_c^{spec} =< G_c^{spec}, M_c, P_c, D_c >$$

Grid Resource Model We consider the grid to be a set of resources with certain capabilities and policies. The grid can be modeled as a weighted resource graph where the nodes represent computational resources and the edges represent the network connectivity between the resources. Similar to application entities, grid resources have

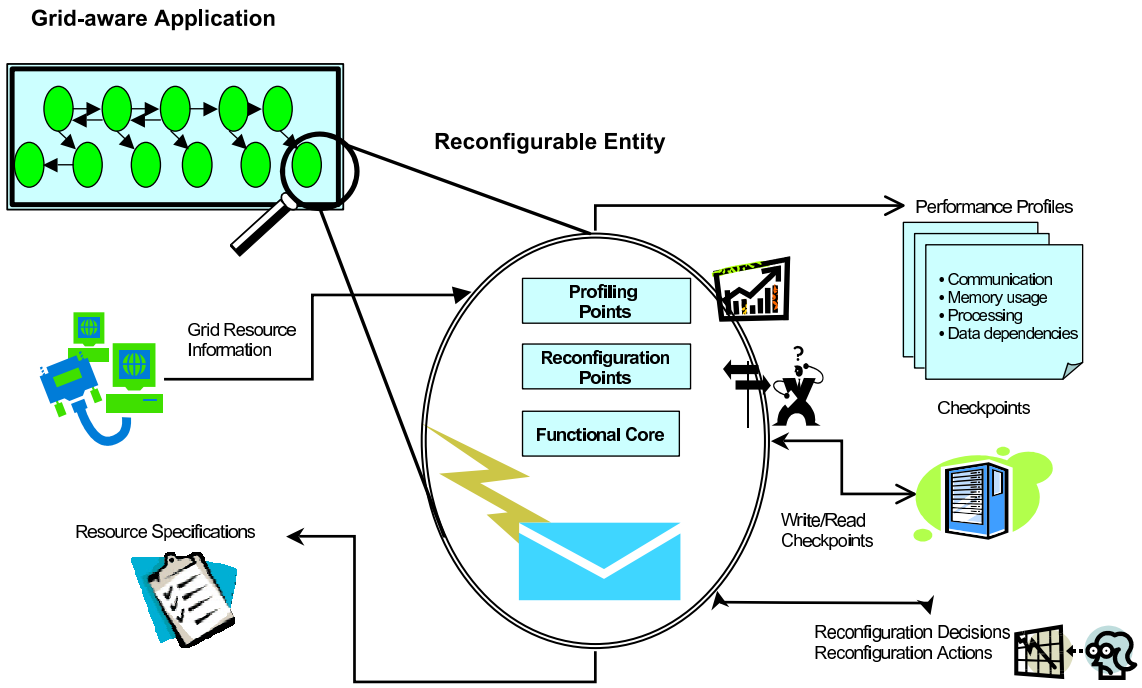

Figure 3.3: Model for a grid-aware application.

dynamic performance profiles and static performance profiles. The dynamic profiles represent the current utilization of resources, such as processing rate, communication latency and bandwidth, amount of available memory, and amount of available disk storage. The static profiles represent the static resource characteristics such as hardware platforms, CPU speed, number of processors, memory capacity, network latency, and network bandwidth.

Both grid resources and application entities need to monitor their performance progress. At the level of the application, entity-level profiling is used to monitor periodically the performance of processing, communication, and data access. This is what generates entity performance profiles. We call the stages where profiling is triggered while the entity is running, *profiling points*. The *functional core* refers to the useful work the entity is doing such as processing and communication. Figure 3.3 illustrates the overall model of a grid-aware application and the structure of its entities.

Reconfigurable Entity States Application entities can be in one of these states:

- **Running:** Either computing, communicating, or profiling its performance.
- **Attempting reconfiguration:** This state happens only at a reconfiguration point. The middleware tries to reason about the current state of the entity and the state of the available grid resources. A reconfiguration happens if it is expected that the application will be transformed into a better configuration.
- **Checkpointing:** If the system decides to reconfigure a specific entity, the state of the entity is saved to prepare for migration.
- **Splitting:** The entity is splitting into more entities.
- **Merging:** The entity is joining other entities to form one entity of larger granularity.
- **Migrating:** Migration to another grid resource. This step involves migrating the checkpointed state of the entities and any binaries or data files it needs.
- **Restarting:** Once the entity is received by the target resource, it is resumed by reading its checkpoints and reconstructing its state.

Figure 3.4 shows a flowchart of the possible states of a reconfigurable entity and the decisions involved in this process.

3.3 IOS Middleware Architecture

An IOS-enabled environment is a set of agents that forms a virtual network. An IOS agent is implemented as a SALSA actor and therefore inherits all the autonomous features of actors. Additionally the use of SALSA and Java makes IOS a portable framework. Although IOS has been implemented in SALSA and Java, it is not limited to reconfiguring only SALSA programs. Indeed, we demonstrate in Chapter 5 how iterative MPI applications have been successfully reconfigured using the IOS middleware.

Every IOS agent consists of three pluggable modules that enable evaluation of different types of autonomous reconfiguration, and also allow users to develop their

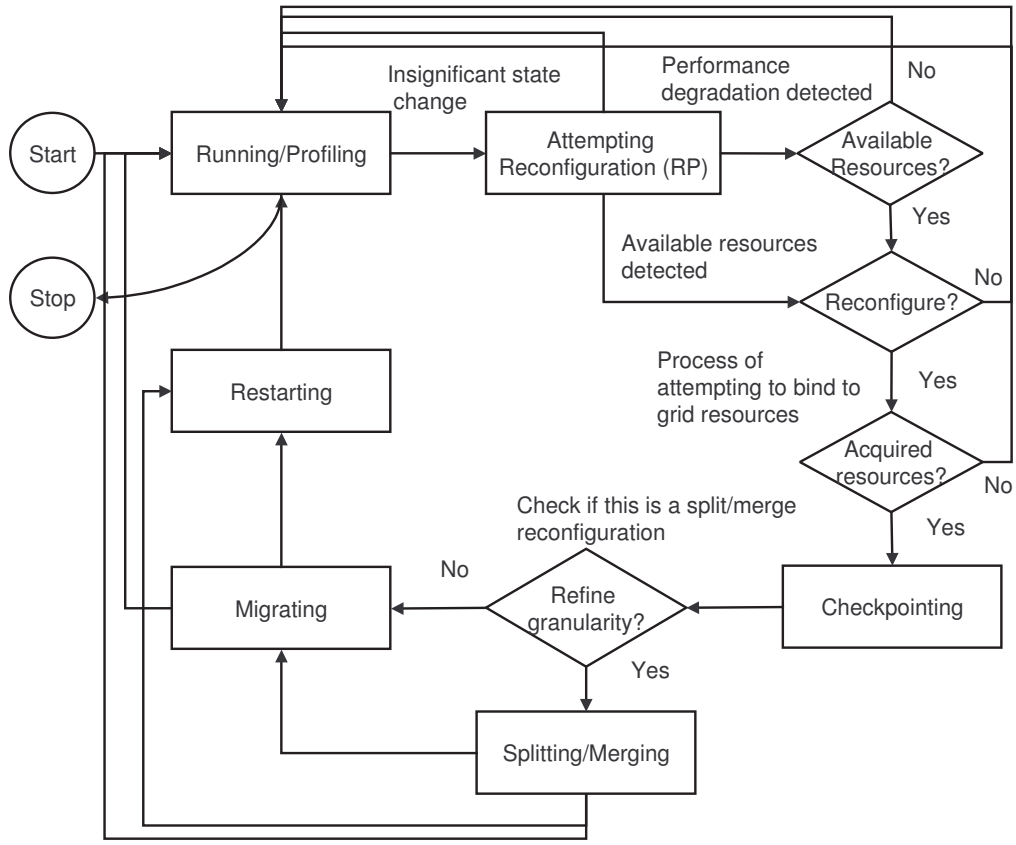

Figure 3.4: The possible states of a reconfigurable entity.

own fine-tuned modules for particular applications. These are the profiling module, the decision module, and the protocol module (see Figure 3.5).

3.3.1 The Profiling Module

Resource-level and application-level profiling are used to gather dynamic performance profiles about physical resources and application entities (such as processes, objects, actors, or agents). Application entities communicate processing, communication, data access and memory usage profiles to the middleware via a profiling API. Profiling monitors periodically measure and collect information about resources such as CPU power, memory, disk storage, and network bandwidth. The IOS architecture defines interfaces for profiling monitors to allow the use of various profiling technologies. Examples include the Network Weather Service (NWS) [118] and Globus Meta

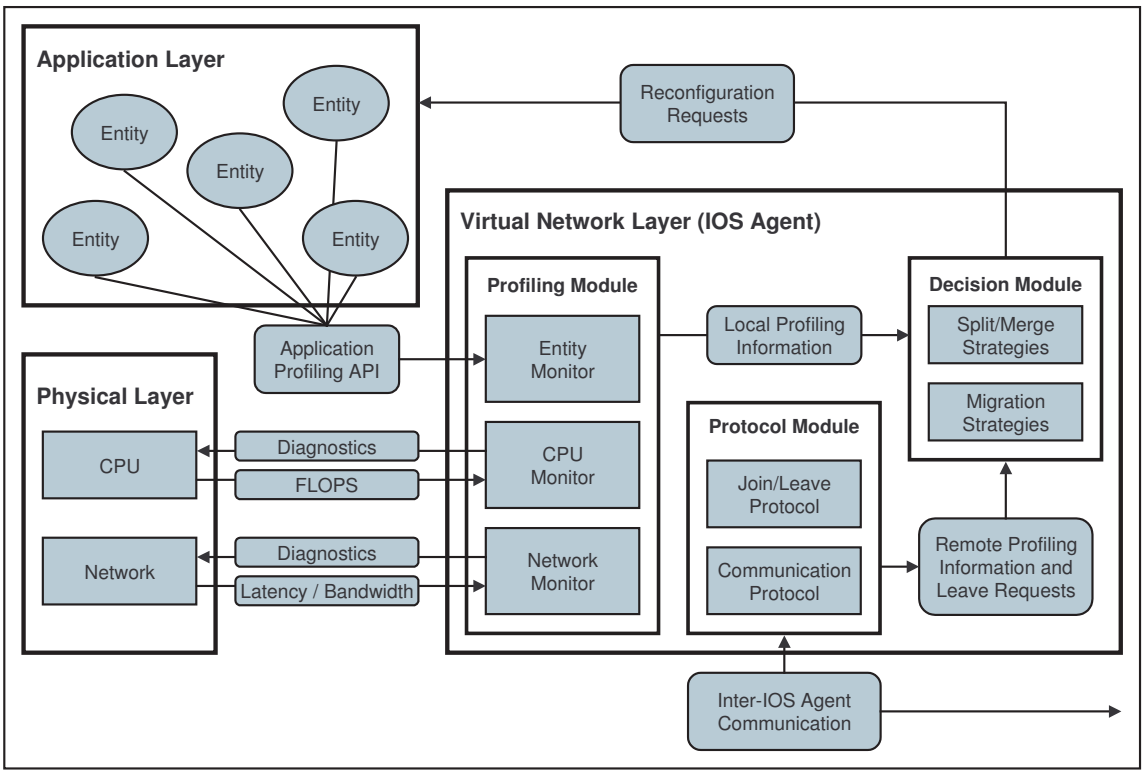

Figure 3.5: IOS Agents consist of three modules: a profiling module, a protocol module and a decision module. The profiling module gathers performance profiles about the entities executing locally, as well as the underlying hardware. The protocol module gathers information from other agents. The decision module takes local and remote information and uses it to decide how the application entities should be reconfigured.

Discovery Service [36]. The profiling API is described in more details in Section 3.4.1. Figure 3.6 shows the architecture of the profiling module and how it interfaces with high-level applications and local resource monitors. The module generate dynamic performance profiles pertaining to application entities and local resources. These performance profiles are fed to the decision module to make informed decisions about a potential reconfiguration.

To illustrate an example of interfacing with an existing monitoring tool. We have used the NWS monitoring system to periodically gather machine performance profiles. NWS [118] has been designed to monitor dynamic system information. It

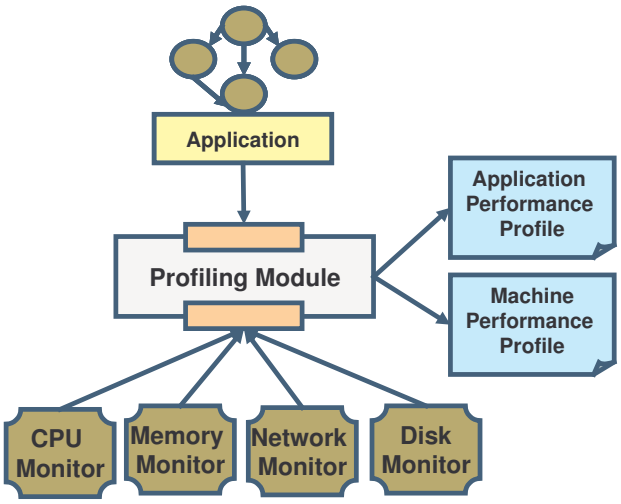

Figure 3.6: Architecture of the profiling module: this module interfaces with high-level applications and with local resources and generates application performance profiles and machine performance profiles.

provides real-time estimation of system performance as well as a forecasting of resource availability in the next 10 seconds. NWS also conducts end-to-end network probes to measure the avalability of network-related performance. NWS supports online monitoring of the following system attributes:

- **available CPU:** the fraction of the available CPU,
- **current CPU:** the fraction of the CPU already used,
- **free memory:** the amount of free memory,
- **free disk:** the amount of free disk,
- **connectTimeTcp:** the time required to establish a TCP connection to a remote host,
- **bandwidthTcp:** the speed of sending data to a remote host, and
- **latencyTcp:** the time it takes to send an empty TCP message to a remote host.

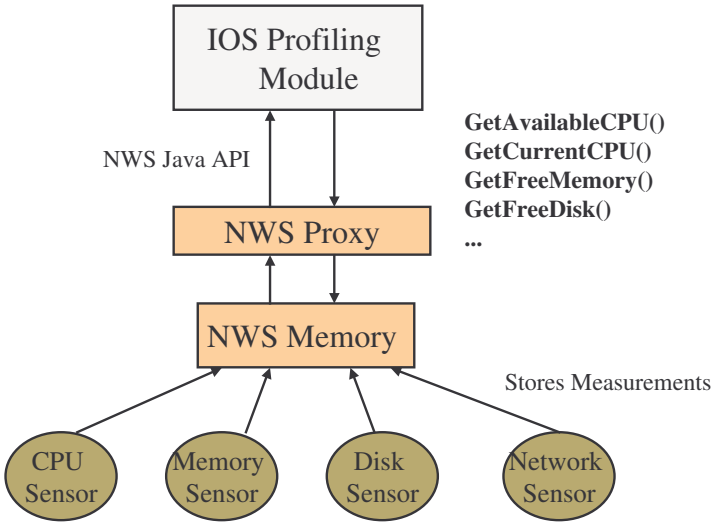

Figure 3.7: Interactions between the profiling module and the Network Weather Service (NWS) components.

An NWS system consists typically of 1) a name server that is responsible for maintaining the list of hosts that are being monitored and the time-series that are being gathered for them, 2) a memory component responsible for remotely storing measurements, 3) sensors, which have to be started on every host that needs to be monitored. Sensors take measurements and report them to the memory component, 5) a forecaster, which analyzes a series of measurements and predicts using statistical techniques the availability of the resources in the near future and, 4) a proxy, which aggregates multiple measurements and reports the last forecast values. Figure 3.6 shows the interactions of the profiling module with the NWS components. We have used the Java API of the NWS to interface with the proxy and get the last forecast values of CPU, memory, network, and disk availabilities.

3.3.2 The Decision Module

Using locally and remotely profiled information, the decision module determines how reconfiguration should be accomplished. Different reconfiguration strategies such as random work-stealing, application topology sensitive work-stealing and network topology sensitive work-stealing have been implemented (refer to Chapter 4 for a detailed explanation of these strategies).

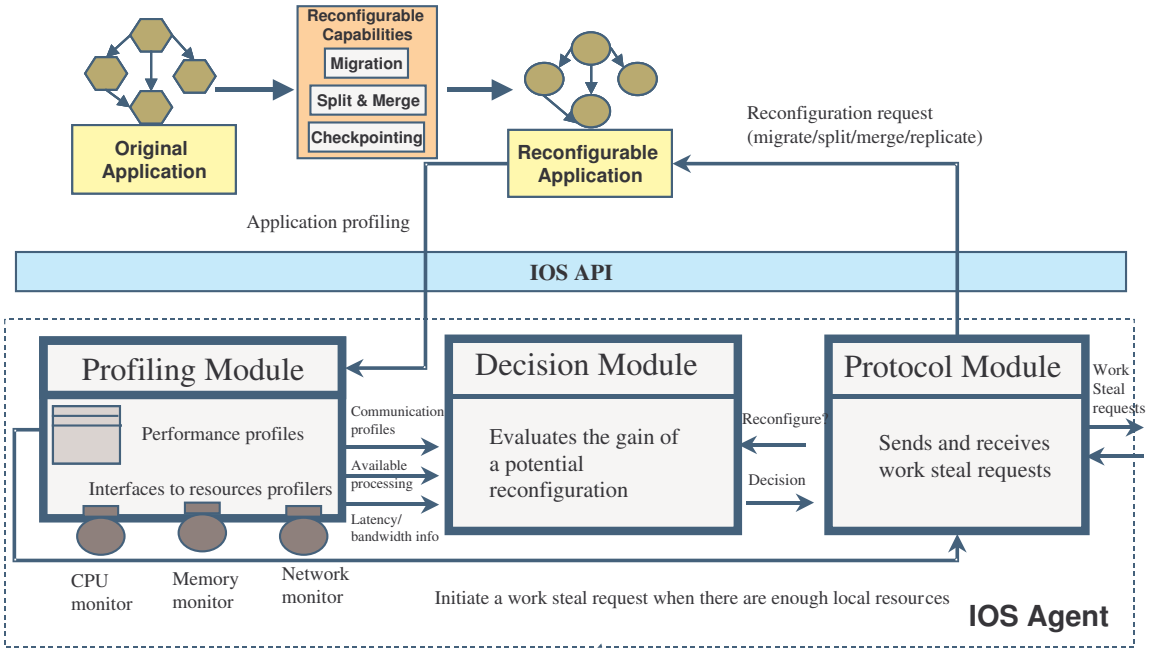

Figure 3.8: Interactions between a reconfigurable application and the local IOS agent.

3.3.3 The Protocol Module

The protocol module is responsible for inter-agent communication. It is used to distribute profiled information among the different middleware agents. The agents can connect themselves into various virtual agent topologies forming a *virtual network*. Examples include peer-to-peer and hierarchical interconnections. All protocols implemented as part of the protocol module should follow the work-stealing model and its variations (see Section 4.2). In this model, idle nodes request work from peer nodes. This module takes care of initiating work steal requests from the local node and handling the reception of work steal requests from remote nodes.

Figure 3.8 shows the interactions between the IOS modules, and the interactions between the IOS agents and the application's entities. As mentioned before, the profiling module periodically gathers performance profiles about the application's entities and the underlying resources hosted locally. The profiling module maintains a list of performance profiles and resource profiles and communicates them to the decision module. When a work-steal request is received by the protocol module, it contacts the decision module, which determines based on both the characteristics

of the remote node and the gathered performance profiles whether an application reconfiguration should happen or not. Any reconfiguration is then communicated to the application via the protocol module through the IOS API.

3.4 The Reconfiguration and Profiling Interfaces

The application profiling and reconfiguration request APIs in IOS are generic. To illustrate their generic nature, these APIs have been implemented for two different programming models: actors, using the SALSA programming language, and communicating processes, using MPI. Our choice of these two programming models has been influenced by the fact that SALSA is a programming language with several embedded features for reconfiguration such as migration and asynchronous message passing while MPI is a standard model that has gained a widespread usage among the parallel processing community.

The following sections discuss IOS generic APIs and give a sample implementation using SALSA. The MPI implementation is discussed in more detail in Chapter 5.

3.4.1 The Profiling API

Figure 3.9 shows the methods of the profiling API with their arguments. The profiling API provides a set of methods for recording the communication patterns of the application entities. The methods `addProfile()`, `removeProfile()`, and `migrateProfile()` are used to track the entry and exit of entities in and out of the local node. They are used to create or delete performance profiles for the locally hosted entities. The methods `msgSend()` and `msgReceive()` are used to profile the communication between any couple of entities. Finally the `beginProcessing()` and `endProcessing()` methods are used to record the time spent processing a received message.

Note that all entities are regarded by the IOS middleware as actors. Therefore the terminology used in the API pertains to actors. Every application entity should be assigned a unique name that conforms to the UAN naming scheme [113]. Entities also have a UAL that identifies their current location. Some of the API methods take the entity's UAN and UAL as arguments.

```
public interface ProfilingAgent {


//The following methods notify the profiling agent of entities
//entering and exiting the local run-time system due
//to migration or initial entity startup.

        public void addProfile(UAN uan);
        public void removeProfile(UAN uan);
        public void migrateProfile(UAN uan, UAL target);

//The profiling agent updates its entity profiles based
//on message sending with these methods

   public void msgSend(UAN uan,
                       UAL target_uan,
                       UAL source_ual,
                        int msg_size);

//The profiling agent updates its entity profiles based
//on message reception with this method

   public void msgReceive(UAN uan,
                          UAL target_ual,
                          UAL source_ual,
                           int msg_size);

//The following methods notify the profiling agent of the start
//of a message being processed and the end of a message
//being processed

   public void beginProcessing(UAN uan,
                               UAL target_ual,
                               UAL source_ual,
                                int msg_size);

   public void endProcessing(UAN uan,
                             UAL target_ual,
                             UAL source_ual,
                              int msg_size);

}
```

Figure 3.9: IOS Profiling API

3.4.2 The Reconfiguration Decision API

Figure 3.10 shows the API used to decide the candidate entities for a reconfiguration and also when a reconfiguration should happen.

When a remote node attempts to steal work, it sends its performance profile and the collection of performance profiles of the local nodes it is hosting to one of its peer nodes. This information, together with the information gathered about the local profiles, is used to evaluate if a reconfiguration though migration, split, or merge should happen to the target machine. The `decisionFunction()` method returns a real value that indicates the expected gain that could be achieved from reconfiguring the entity with performance profile `candidate`. The parameters that the `decisionFunction()` API call takes are: 1) `candidate`, which denotes the performance profile of an entity, 2) `target`, the UAL or location of the remote machine that is the target for a potential migration, split, or merge operation, 3) `targetMachine`, the performance profile of the remote machine, and 4) `actorsAtTarget`, which denotes a vector of the performance profiles of the entities hosted at the remote machine.

The `getBestCandidate()` API call returns the best candidate for a potential reconfiguration. The candidates are ranked based on the expected gain value returned by the `decisionFunction()`. The method takes as parameters: 1) `target`, the location of the remote machine, 2) `targetMachine`, the performance profile of the remote machine, and 3) `actorsAtTarget`, the performance profiles of the group of actors hosted at the remote machine. `getBestCandidates()` is similar to `getBestCandidate()`. The only difference is that it returns a group of the best candidate actors with size `group_size`.

The `getLoadDistribution()` API call returns a vector of two numbers that indicates the new distribution of the applications entities running in the local and remote machine. The first number in the vector denotes the number of candidates to be transferred to the remote machine, while the second number indicates the number of entities to remain in the local machine. The method takes as parameters: 1) `localMachine`, the performance profile of the local machine, 2) `targetMachine`, the performance profile of the remote machine, 3) `actorsAtLocal`, a vector of the performances profiles of the actors hosted locally, and 4) `actorsAtTarget`, a vector of

the performance profiles of the actors hosted remotely. Let N_{old} be the total number of entities running in the local machine before reconfiguration, let n_{local} and n_{remote} denote the resulting number of entities in the local machine and remote machines respectively after reconfiguration. Merging is a purely local decision that is not triggered by a remote request for reconfiguration. The `getLoadDistribution()` API call only applies to migrate and split reconfigurations. Therefore, the inequality 3.1 should always hold after the invocation of `getLoadDistribution()` method.

$$N_{old} \leq n_{local} + n_{remote} \tag{3.1}$$

When $N_{old} = n_{local} + n_{remote}$, the resulting reconfiguration is a migration operation. When $N_{old} < n_{local} + n_{remote}$, the resulting reconfiguration involves both split and migration operations.

3.5 Case Study: Reconfigurable SALSA Actors

Figure 3.11 shows the UML diagram of some the key classes that implement a SALSA actor and the extensions that were made to make SALSA actors reconfigurable through IOS. Every SALSA actor has a state and a reference. The state encapsulates the actor's internal message handlers and variables. The actor also has a mailbox that maintains all the messages received. The `Message` class is an abstraction over a message being sent to an actor. It includes a field of type `java.lang.reflect.Method` and a set of arguments. This method will eventually be invoked with the given arguments when the enclosing message is received and processed at the target actor. The `UniversalActor` class provides the main functionalities for universal naming though UAN and UAL mapping, and mobility through migration.

To make SALSA actors reconfigurable, the `AutonomousActor` behavior extends `UniversalActor` with transparent profiling of its behavior and integration with IOS middleware. Any SALSA actor that extends the `AutonomousActor` behavior becomes capable of autonomous migration through IOS. The `AutonomousActor` behavior overrides most of the `UniversalActor` methods that pertain to communication, and message processing, and sends related events to the IOS profiling agent. To provide extensions for malleability, or split and merge capabilities, the `MalleableActor`

behavior extends further the `AutonomousActor` behavior and provides a reference implementation for handling malleability messages. The `MalleableActor` is an abstract behavior. It provides the API specification for split and merge messages. Actor malleability is application dependent and the proper implementation of the split and merge functionalities must be provided by the application programmer. Both split and merge message handlers take as arguments a vector of actors to be involved in the split or merge operations and the size of the new vector of actors after malleability takes place, in other words, it indirectly specifies how many more actors we need to split, or how many actors we need to merge.

3.6 Chapter Summary

We presented in this chapter the software framework of the Internet Operating System middleware and its architecture. We started by discussing several design decisions that were taken into consideration while designing and implementing IOS. IOS middleware agents: 1) profile application communication patterns, 2) evaluate the dynamics of the underlying physical resources, and 3) reconfigure application entities by changing their mappings to physical resources through migration and/or changing their granularity through split and merge operations. IOS exposes generic APIs for profiling and reconfiguration that can easily hook up various profiling and reconfiguration implementations to the middleware. This makes it programming model-independent: we have implemented an actor programming model interface for SALSA programs and also a process programming model interface for MPI programs. The chapter concludes with a case study showing the extensions that were made to the SALSA actor library to add reconfigurability features.

```
// The following method returns a real value
// that evaluates the expected gain of reconfiguring
// the actor candidate to the machine target.

double decisionFunction(ActorProfile    candidate,
                        UAL             target,
                        MachineProfile  targetMachine,
                        Vector          actorsAtTarget);

// The following method returns a reference to the best candidate
// for reconfiguration

ActorReference getBestCandidate(UAL             target,
                                MachineProfile  targetMachine,
                                Vector          actorsAtTarget);

// The following method returns a group of references
// for reconfiguration

Vector getBestCandidates(UAL             target,
                         MachineProfile  targetMachine,
                         Vector          actorsAtTarget,
                         int             group_size);

// The following method returns the size of the actors
// to be transferred to the target machine to achieve
// load balance between the local and target machines

Vector getLoadDistribution(MachineProfile localMachine,
                           MachineProfile targetMachine,
                           Vector         actorsAtTarget,
                           Vector         actorsAtLocal);
```

Figure 3.10: IOS Reconfiguration API

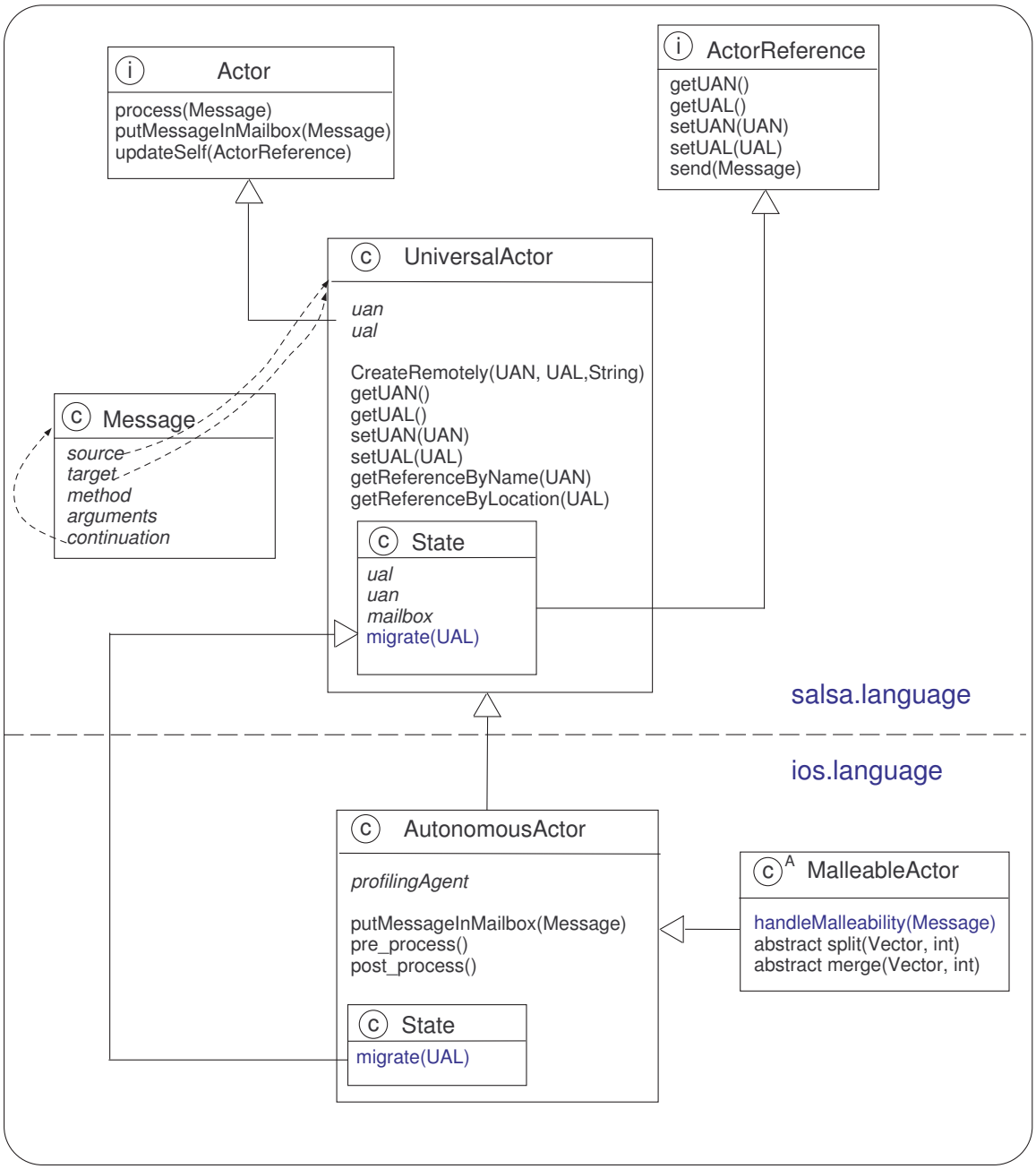

Figure 3.11: A UML class diagram of the main SALSA/IOS Actor classes and behaviors. The diagram shows the relationships between the Actor, UniversalActor, AutonomousActor, and MalleableActor classes.

CHAPTER 4

Reconfiguration Protocols and Policies

In the previous chapter, we gave an overview about the IOS framework and its main components. This chapter discusses the resource discovery mechanisms and reconfiguration protocols that have been adopted as part of IOS. Resource discovery and dissemination in IOS are based on peer-to-peer (P2P) protocols. IOS agents interact based on their virtual interconnection topology. Section 4.1 presents how IOS agents interconnect with each other and exchange information. To balance computational loads in an IOS virtual network, we introduce in Section 4.2 variations of the classical work-stealing algorithm: random work-stealing (RS), application-topology sensitive work-stealing (ATS) and network-topology sensitive work-stealing (NTS). We discuss in this section the load balancing policies used: the information exchange policy, the transfer policy, and the peer location policy. In Section 4.3 we present the selection policy used. Split and merge policies are discussed in Section 4.4. Related work is presented in Section 4.5. The chapter concludes with a summary in Section 4.6.

4.1 Network-Sensitive Virtual Topologies

The IOS architecture has a decentralized architecture to ensure robustness, scalability, and efficiency. Each IOS-ready node is equipped with a middleware agent. Agents organize themselves in various virtual network topologies to sense the underlying physical environment and trigger applications' reconfiguration accordingly. Every IOS agent also has a decision component that evaluates the surrounding environment

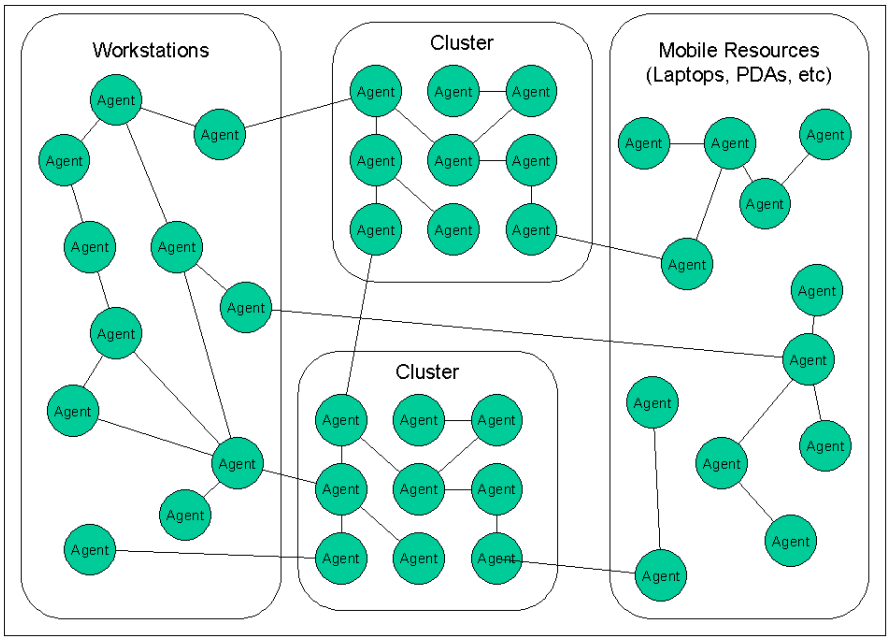

Figure 4.1: The peer-to-peer virtual network topology. Middleware agents represent heterogeneous nodes, and communicate with groups or peer agents. Information is propagated through the virtual network via these communication links.

and decides how to balance the resource consumption of application's entities in the physical layer.

We call a *network sensitive* virtual topology, any topology that adjusts itself according to the underlying network topologies and conditions. Two types of topologies are used in IOS: a *peer-to-peer* (P2P) topology and a *cluster-to-cluster* (C2C) topology. The P2P topology consists of several heterogeneous nodes inter-connected in a peer-to-peer fashion while the c2c topology imposes more structure on the virtual network by grouping homogeneous nodes with low inter-network latencies into clusters.

4.1.1 The Peer-to-Peer Topology

When an agent connects to the network-sensitive peer-to-peer (NSp2p) virtual network, it can discover new peers as information gets passed across peers. Agents can also dynamically leave the virtual network. Every peer maintains a list of neighbors. This list can grow or shrink as information is propagated across the virtual network.

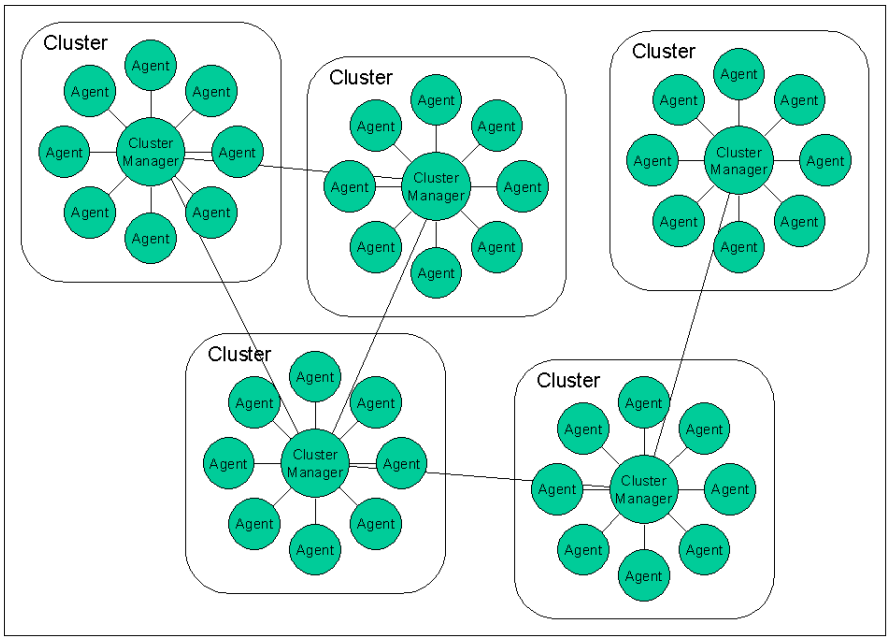

Figure 4.2: The cluster-to-cluster virtual network topology. Homogeneous agents elect a cluster manager to perform intra and inter cluster load balancing. Clusters are dynamically created and readjusted as agents join and leave the virtual network.

Section 4.2 describes how resource information gets passed between the peers.

4.1.2 The Cluster-to-Cluster Topology

In the network-sensitive cluster-to-cluster topology (NSc2c), agents are organized into groups of virtual clusters (VCs), as shown in Figure 4.2. Each VC elects one agent to act as the cluster manager. Cluster managers view each other as peers and organize themselves as a NSp2p virtual network topology.

4.1.3 Presence Management

To participate in the IOS virtual network, a peer must first know an already existing IOS agent. Specialized servers exist for this purpose, called *peer servers*. Upon contacting a peer server, an agent registers itself and gets a list of well-known peers as a bootstrapping mechanism. Each peer maintains a list of neighboring peers. Peer servers act as registries for agent discovery. They simply aid in discovering peers in a virtual network and are not a single point of failure. They operate similarly to

`gnutella-hosts` in Gnutella peer-to-peer networks [33]. A node can bypass the peer server by contacting directly an existing IOS agent, if it knows of such a host. The resulting network is an overlay of peer agents. Figure 4.3 illustrates the mechanism of joining the IOS virtual network. The procedure used for joining the virtual network through a peer server is further illustrated in Figure 4.4.

Upon reception of a request from a peer, the receiving agent adds this peer to its neighborhood if it does not exist. This strategy leads to discovering more peers. The reconfiguration strategies used avoid using broadcasts to minimize the overhead associated with maintaining the virtual network and exchanging reconfiguration information.

When a node decides to leave the virtual network, it sends a leave request to the peer server it initially registered with, if any (see Figure 4.3). It also broadcasts a leave request to all its neighbors. The leaving node is then deleted from the peer server registry and also from the list of neighbors in all of its peers. Any computing entities that are hosted by this node will be migrated immediately to its peers in a round-robin fashion.

4.2 Autonomous Load Balancing Strategies

IOS load balancing strategies are extensions of the classical work-stealing approach. Work-stealing [21] has traditionally been used as a scheduling technique for multi-threaded computations. Work sharing and work-stealing are two scheduling paradigms that have arisen to address the problem of scheduling multi-threaded computations. In work sharing, the scheduler attempts to move threads from processors that generate more threads to idle processors with the goal of distributing the work evenly between processors. In work-stealing, idle processors attempt to steal threads from overloaded processors. The idea of work-stealing is not new. It was used in early research of parallel execution on functional programs [24] and in an implementation of Multilisp [52]. Rudolph et al. [89] analyzed a randomized work-stealing load balancing strategy for independent jobs on a parallel computer. These works and others have established the benefits of the work-stealing strategy with regards to space and communication. This strategy has been proved to be stable since it minimizes

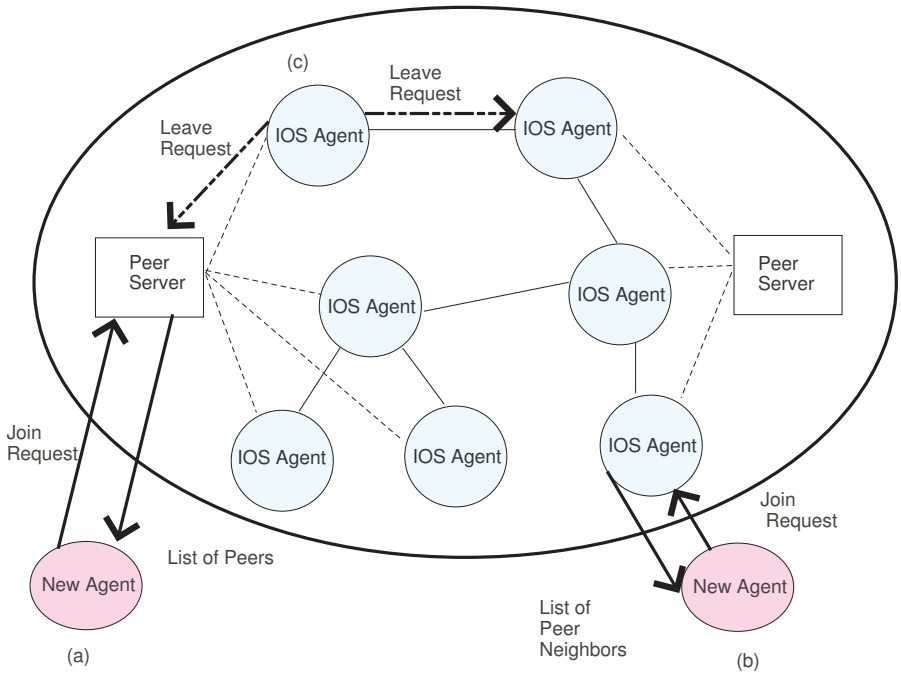

Figure 4.3: Scenarios of joining and leaving the IOS virtual network: (a) A node joins the virtual network through a peer server, (b) A node joins the virtual network through an existing peer, (c) A node leaves the virtual network.

Algorithm 1: Peers Discovery

1: Contact the Peer Server and get a set of peers, $PSet$
2: For all peers $p_j \in PSet$, calculate $d(p_i, p_j)$
/*Distance in terms of network latency*/
3: Select the peer, p_{min} with minimum distance to p_i
4: Contact p_{min} to request its list of peers
5: $S(i) \leftarrow S(p_{min})$
/*p_{min} sends its list of peers to p_i*/
6: $S(p_{min}) \leftarrow S(p_{min}) \cup \{p_i\}$
/*p_{min} includes p_i in its list of peers*/

Figure 4.4: Algorithm for joining an existing virtual network and finding peer nodes.

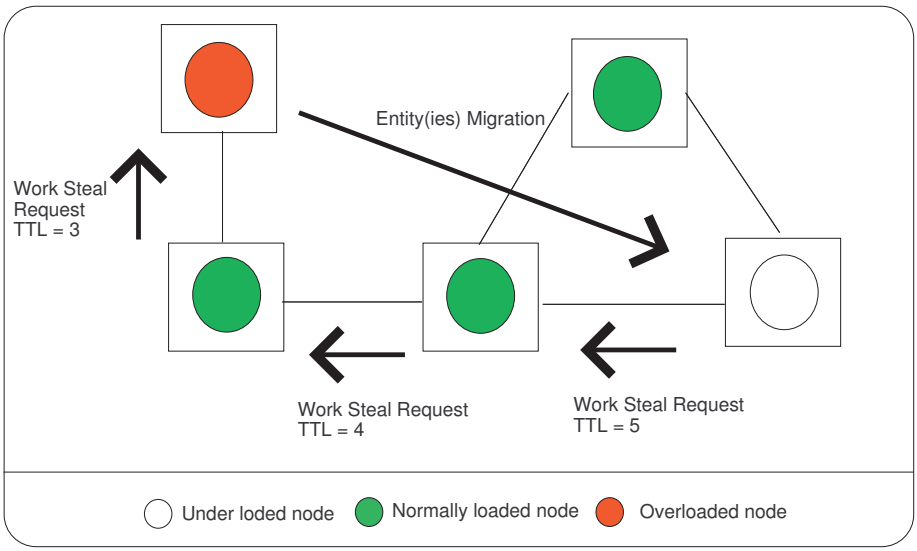

Figure 4.5: An example that shows the propagation of work-stealing packets across the peers until an overloaded node is reached. The example shows the request starting with a time-to-live (TTL) of 5. The TTL is decremented by each forwarding node until it reaches the value of 0, then the packet is no longer propagated.

communication overhead at the expense of idle time [21].

We discuss four basic components of IOS load balancing strategies: an *information policy*, that determines how the state of the peer agents gets collected and exchanged, a *transfer policy*, that determines when a node should participate in transfer of work, a peer *location policy* that determines which peer agent needs to be selected to perform a reconfiguration, and a *selection policy* that decides which entities need to be selected and migrated. We present in what follows the information, transfer, and location policies. The selection policy is based on the resource sensitive model and is presented in Section 4.3. We also show how these policies have been used in the p2p and c2c virtual topologies.

4.2.1 Information Policy

IOS agents exchange performance-related information through work-stealing request messages (WRM). Whenever an agent becomes *lightly loaded* (has more resources available than are currently being utilized), it will periodically send a WRM

containing locally profiled information to one of its peers (see Figure 4.5 for an example). Let TTL_{max} be the upper value for the time-to-live assigned to a given WRM message. WRM messages get forwarded from one peer to another until they reach an agent that has load to be migrated to the requesting agent or the TTL value becomes zero.

Let MP_i denote the machine performance profile of agent a_i. Let $APSet_i$ denote the set of performance profiles of the application's entities running at node i. Let $PSet_i$ be the set of peers of agent a_i. Let TS_i be the time stamp of the measurements in MP_i and $APSet_i$ for agent a_i. Let APW_i denote the available processing power of agent a_i. An agent is considered lightly loaded if its APW_i is above a predefined threshold value τ.

Every agent a_i maintains state information about machine performance profiles of itself and its peers in an object called *performance vector*, PV. The PV object allows the agent to have an estimate of the state of its neighborhood, how light or overloaded is its surrounding environment, and how frequent is the change in its surrounding environment. For every agent a_i, the PV object maintains two elements:

- $PV(a_i).MP$: the machine profile of agent a_i
- $PV(a_i).TS$: the time stamp (local time to agent a_i) when the measurements in $PV(a_i).MP$ were taken. We assume that the local clocks in peers are synchronized.

To avoid overloading the network with messages exchanged between peers to get peer state information, state information is piggybacked in the WRM messages. At the beginning, when the agents start, the PV vector has only one element that corresponds to the local agent. Once an agent receives WRM messages, it increases its local vector by inserting state information about the sender. This strategy allows WRM messages to carry state information about its original sender and also about all peers that the WRM message visits. Any time some state information is received through a WRM message, the receiving machine updates its locally stored information in case the information carried in the WRM message is more up-to-date. This is done by comparing the timestamps of the measurements. Figure 4.6 shows the algorithm

used for information exchange between peer IOS agents and how state information get propagated across the peers.

4.2.2 Transfer Policy

The purpose of the transfer policy is to determine when to transfer load from one agent to another and how much load needs to be transfered, whenever a WRM message is sent from an agent a_i to an agent a_j. We identify the load of a given agent simply by the number of running application' entities hosted by this agent. We denote by Nb_i the number of application's entities running on agent a_i before a given reconfiguration step, and by Na_i the number of application's entities running on agent a_i after a given reconfiguration step. Let Na_i and Na_j refer to the number of applications entities running on agents a_i and a_j respectively. Let APW_i be the percentage of the available CPU processing power of agent a_i and UPW_j be the percentage of the used CPU processing power of agent a_j. Let PW_i and PW_j be the current processing powers of agents a_i and a_j respectively. The transfer policy tries to adjust the load between two peer agents based on their relative machine performances as shown in the equations below:

$$N_{total} = Nb_i + Nb_j \tag{4.1}$$

$$\frac{Na_i}{APW_i * PW_i} = \frac{Na_j}{UPW_j * PW_j} \tag{4.2}$$

$$Na_i = \frac{APW_i * PW_i}{APW_i * PW_i + UPW_j * PW_j} * N_{total} \tag{4.3}$$

$$Na_j = \frac{UPW_j * PW_j}{APW_i * PW_i + UPW_j * PW_j} * N_{total} \tag{4.4}$$

$$N_{transfer} = Na_j - Nb_j \tag{4.5}$$

This equation allows us to calculate the number of entities that need to be transfered to remote agent a_i to achieve load balance between a_i and a_j (see Equation 4.5). All the entities in host a_j are ranked according to a heuristic decision function that calculates their expected gain from moving from agent a_i to agent a_j. The details about how the gain value is calculated are given in Section 4.3. Only the entities

Algorithm 2: Information Exchange Policy

At the initiating agent a_i

1: **if** $APW_i > \tau$ **then**
2: $a_j \leftarrow SelectPeer(PSet_i, locationPolicy)$
3: $TTL \leftarrow TTL_{max}$
4: $PV(a_i).TS \leftarrow TS_i$
5: $PV(a_i).MP \leftarrow MP_i$
6: Append ($PV(a_i).TS$, $PV(a_i).MP$, $APSet_i$, TTL) to the work-stealing message request, WRM
7: Send message WRM to agent a_j
8: **end if**

At a receiving agent a_j

1: **if** $a_i \notin PSet_j$ **then**
2: $PSet_j \leftarrow PSet_j \cup \{a_i\}$
3: **end if**
/*X is the set of all agents whose information is carried in the WRM message*/
4: **for all** $a_x \in X$ **do**
5: **if** ($a_x \in PSet_j$) and ($PV(a_x).TS > PV_j(a_x).TS$) **then**
6: $PV_j(a_x).MP \leftarrow PV(a_x).MP$ /*Update the sate of the neighbor using the information carried in the WRM message*/
7: **end if**
8: **end for**
9: **if** (There are no entities to be transfered according to the *transfer policy*) **then**
10: $TTL \leftarrow TTL - 1$ /*Decrement the value of TTL and update it in the WRM message*/
11: **if** ($TTL > 0$) **then**
12: $PV(a_j).TS \leftarrow TS_j$
13: $PV(a_j).MP \leftarrow MP_j$
14: Append ($PV(a_j).TS$, $PV(a_j).MP$) to the work-stealing message request WRM. /*Append the machine performance profile of this agent to WRM*/
15: $a_k \leftarrow SelectPeer(PSet_j, locationPolicy)$
16: Forward message WRM to agent a_k
17: **else**
18: Notify the source agent a_i that its WRM message is no longer forwarded
19: **end if**
20: **else**
21: select the set of entities $ESet$ to be transfered according to the *transfer policy*
22: send $ESet$ to a_j
23: **end if**

Figure 4.6: Information exchange between peer agents using work-stealing request messages.

Group Classification	Range of Latencies
Local	0 to 10 ms
Regional	11 to 100 ms
National	101 to 250 ms
Global	251 ms and higher

Table 4.1: The range of communication latencies to group the list of peer hosts (From [67]).

that have a gain value greater than a threshold value θ are allowed to migrate to the remote agent. So the number of entities that will migrate can be less than $N_{transfer}$. The goal of the gain value is to select the candidate entities that benefit the most from migration to the remote host.

4.2.3 Peer Location Policy

The first policy is random work-stealing (RS) policy. RS is based on a simple protocol: any time a processor senses that it is idle, it attempts to steal some jobs from randomly selected peers. However in grid environments, using RS tends to incur a lot of WAN latencies since the peers are treated equally. We use variations of the RS approach to account for WAN latencies and for grid characteristics.

The second policy tries to account for the wide range latencies that are predominant in grid environments. Peers are sorted according to their latency. The algorithm picks the closest peer first, then the next closer and so on. Every agent arranges its peers into four groups based on communication latency [67]: local, regional, national, and global. Table 4.2.3 shows the categorization of peers according to their latencies.

WRMs are sent first locally. When no migration happens, the source of the WRM will send another WRM to a regional peer, and if no migration occurs again, a WRM is sent nationally, then globally. As reconfiguration is only triggered by lightly loaded nodes, no overhead is incurred when the network is fully loaded, and thus this approach remains stable.

4.2.4 Load Balancing and the Virtual Topology

All the strategies discussed before refer to load balancing in the case of the peer-to-peer virtual topology. Load balancing in the NSc2c topology differs from the NSp2p topology mainly in the granularity of the peer agent. A peer in the NSc2c case is a cluster. So the peer in this case has a larger granularity. When a cluster becomes idle, then it starts prompting its peer clusters for work. The same policies apply in the NSc2c case.

The cluster-to-cluster strategy attempts to utilize central coordination within VCs in order to obtain an overall picture of the applications' communication patterns and resource consumption as well as the physical network of the VC. A cluster manager acts as the central coordinator for a VC and utilizes this relatively global information to provide both intra- and inter-VC reconfiguration.

Every cluster manager sends periodic profiling requests to the agents in its respective VC. Every agent responds with information from its profiling component about the local entities and their resource consumption. The cluster manager uses this information to determine which entities should be migrated from the node with the least available resources to the node with the most available resources.

Let n_1 and n_2 be the number of entities running on two nodes, and $r_{i,j}$ be the availability of resource i on node j with a resource weight w_i. The intra-cluster load balancing continuously attempts to achieve the relative equality of application's entities on nodes according to their relative resource availability (Equation 4.6):

$$\frac{n_1}{n_2} = \frac{\sum w_i r_{i,1}}{\sum w_i r_{i,2}} \tag{4.6}$$

For inter-cluster load balancing, NSc2c uses the same strategy as peer-to-peer load balancing, except that each cluster manager is seen as a peer in the network. The cluster managers decision component compares the heaviest loaded node to the lightest loaded node at the source of the WRM to determine which entities to migrate.

4.3 The Selection Policy

4.3.1 The Resource Sensitive Model

The reconfiguration policies use the application's characteristics and the underlying resources' characteristics to evaluate whether there will be any performance gain through migration. Resources such as storage, CPU processing power, network bandwidth, and network latencies are ranked according to their importance to the running application. For instance, if the application is computationally intensive, more weight is given to the CPU processing power. If the application is communication intensive and the messages exchanged have large sizes, more weight is given to the network latency and bandwidth. Whereas, if it is communication intensive with small exchanged message sizes, the network latency is given more weight.

Let P be the set of m processes running on a local node n.

$$P = \{p_0, p_1, ..., p_m\}$$

Let R be the set of resources available in the local node n.

$$R = \{r_0, r_1, ..., r_l\}$$

Let ω be the weight assigned to a given resource r_i based on its importance to the performance of the set P.

$$0 \leq \omega(r_i, P) \leq 1 \text{ and } \sum_{i=0}^{l} \omega(r_i, P) = 1$$

Let $a(r, f)$ be the amount of resource r available at foreign node f, $u(r, l, A)$ be the amount of resource r used by the processes P at local node l, $M(P, l, f)$ be the estimated cost of migration of the set P from l to f, and $L(P)$ be the expected remaining life expectancy of the set of processes P. Let m be the average number of times the set of processes have migrated in the past. Equation 4.7 measures the normalized benefit, the set of processes P will incur from migration to the remote node l, given ideal conditions.

We introduce the notion of process energy, $E(P)$. This values measures how

much energy the process(es) has to be able to migrate. The larger the energy of a process, the more willing it is to migrate. The energy of the process is inversely proportional to how many times the process has migrated in the past. Equation 4.8 shows the relationship between the energy of a set of processes P and their average number of migrations in the past. The value of $E(P)$ gives an indirect measure of the stability of the surrounding environments. The smaller the value of $E(P)$, the less stable the environment is because the group of entities have migrated frequently in the past. The larger the energy value the more stable the environment is.

The expected gain value Γ, which heuristically measures the increase in overall performance gained by migrating P from l to f, where $\Gamma \leq 1$ is shown in Equation 4.9. It is important to factor in this equation the remaining life expectancy of the group of processes P. The intuition behind this is that processes who are expected to have a short remaining life are not migrated because the cost of the reconfiguration might exceed the benefit over their remaining life expectancy. The energy value is dominated by the remaining life expectancy when the value of this latter is large. The rationale is that long lived processes should still be migrated whenever there is benefit even if they do not have a lot of energy. Because their long expected remaining life amortizes the cost of migration. However for short-lived processes, the energy value will dominate and will decrease the overall benefit of migration.

Figure 4.7 illustrates the behavior of the expected gain function (Equation 4.9). The graph shows plots of the expected gain function for one process with a benefit value of 0.6 ($\Delta_{r,l,f,P} = 0.6$) and different values of m, the number of times the process has has migrated in the past. The remaining lifetime of a process is assumed to have a *half-life* expectancy. In other words, if a process has lived for a period of t, its expected remaining lifetime is $L(t) = t$. For the purpose of illustration, we assume that the cost of migration is constant and it is assumed here to be $1s$. The plots show that when the process is short lived, the energy function affects the gain function, however when the process is long lived, the remaining life expectancy is what dominates the second term of the equation and the gain function has a larger value.

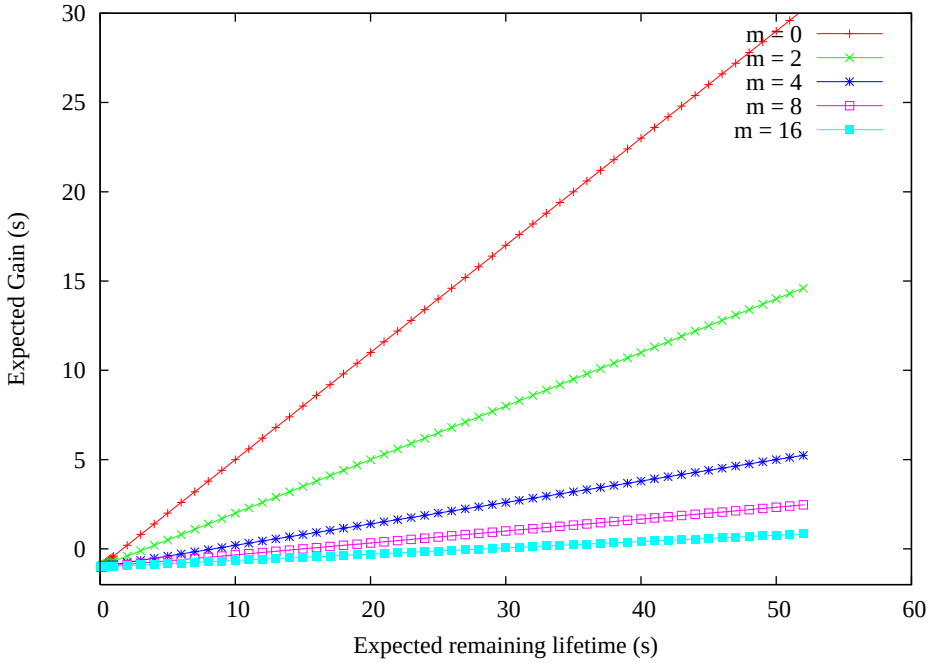

Figure 4.7: Plots of the expected gain decision function versus the process remaining lifetime with different values of the number of migrations in the past. The remaining lifetime is assumed to have a half-life time expectancy.

$$\Delta_{r,l,f,P} = \frac{a(r,f) - u(r,l,P)}{a(r,f) + u(r,l,P)} \tag{4.7}$$

$$E(P) = \frac{1}{log_2(m+1)+1} \tag{4.8}$$

$$\Gamma = (\sum_{r} \omega(r,P) * \Delta_{r,l,f,P}) \times (L(P) * E(P)) - M(P,l,f)) \tag{4.9}$$

When a node l receives a work-stealing request from a node f, a process or a group of processes will be migrated if the evaluation of the gain Γ is greater than a positive threshold value. The larger the gain value is, the more beneficial the migration is.

The purpose of this model is to balance the resource consumption among executing entities. The model has a decentralized approach whereby reconfiguration decisions are done at the level of each grid node. Each entity is profiled to gather

its message sending history with peer entities. This information is used to guide the reconfiguration decision to keep highly communicating entities as closely collocated as possible. The model provides a normalized measure of the improvement in the resource availability an entity or a group or entities would receive by migrating between nodes.

Estimating the expected remaining life of an entity is not usually possible. For the case of iterative applications, we estimate the life expectancy by looking at the number of remaining iterations multiplied by the average time each iteration takes in the current node. When such prediction is not possible like in the case of actor-oriented applications, we use the half-life expected remaining time. That is, if a process has lived for a duration of T seconds, it is expected to have lived half of its lifetime. Therefore, the value of T gives a good estimate of its remaining lifetime. Harchol-Balter and Downey [53] conducted a study to characterize the distribution of lifetimes of UNIX processes. They concluded that UNIX processes have a UBNE (used-better-than-new-in-expectation) type of distribution that is heavy-tailed. In other words, the longer the CPU usage of a process, the longer its expected remaining CPU lifetime. Their results also confirmed the finding of Leland and Ott [71], who also proposed a functional form for the process lifetime distribution. They concluded that for $T > 3s$, such that T is the process duration, the probability of a process's lifetime exceeding T seconds is rT^k, where $-1.25 < k < -1.05$ and r is used to normalize the distribution. Based on these investigations, we conclude that the process half-life measure gives a good estimate about the expected remaining lifetime of a process.

4.3.2 Migration Granularity

The resource model supports both single migration and group migration of application's entities. In single migration, the model is applied to determine an estimation of the gain that would be achieved from migrating an entity from one node to another. If the gain will be achieved by migrating a group of entities, single migration attempts to migrate one entity at a time while group migration strategy will migrate a group of entities simultaneously. One advantage of group migration is that it helps to speed up load balancing. However it might cause thrashing behavior

$$n - l + r = 0 \tag{4.10}$$

$$\frac{APW_r * PW_r}{APW_r * PW_r + UPW_l * PW_l} * n - r = 0 \tag{4.11}$$

$$\frac{UPW_l * PW_l}{APW_r * PW_r + UPW_l * PW_l} * n - l = 0 \tag{4.12}$$

$$n \geq N_{total} \tag{4.13}$$

$$n \in IN^+ \tag{4.14}$$

$$r \in IN^+ \tag{4.15}$$

$$l \in IN^+ \tag{4.16}$$

if the load of the nodes fluctuates very frequently.

4.4 Split and Merge Policies

4.4.1 The Split Policy

The transfer policy discussed in Section 4.2.2 shows how the load in two agents needs to be broken up to reach pair-wise load balance. However, this model will fail when there are not enough entities to make such load adjustments. For example, assume that a local node a_l has only one entity running. Assume also that a_r, a node that is three times faster than a_l, requests some work from a_l. Ideally, we want node a_r to have three time more entities or load than node a_l. However the lack of enough entities prevents such adjustment. To overcome this situation, the entity running in node a_l needs to split into enough entities to send a proportional number remotely.

Let N_{total} be the total number of entities running in both a_l and a_r. Let n be the desired total number of entities, l be the desired number of entities at local node a_l, and r be the desired number of entities at node a_r. APW_r and PW_r denote the percentage of the available processing power and the current processing power of node a_r respectively. UPW_l and PW_l denote the percentage of the used processing power and current processing power of node a_l. The goal is to minimize n subject to constraints shown in the set of equations 4.10 to 4.16 and to solve for n, l, and r in the set of positive natural numbers IN^+.

A split operation happens when $n > N_{total}$ and the entity can be split into one

$$avg_i = \frac{1}{k} * \sum_{t=0}^{k} APW_{i,t} \tag{4.17}$$

$$\sigma_i = \sqrt{\frac{1}{k} * \sum_{t=0}^{k} (APW_{i,t} - avg_i)^2} \tag{4.18}$$

or more entities. In this case the number of entities in local node will be split refining the granularity of the application's entities running in the local node a_l.

4.4.2 The Merge Policy

The merge operation is a local operation. It is triggered when a node has a large number of running entities and the operating system's context switching is large. To avoid a thrashing situation that causes entities to be merged and then split again upon receiving a WRM message, merging happens only when the surrounding environment has been stable for a while. Every peer in the virtual network tries to measure how stable it is and how stable its surrounding environment is.

Let $S_i = (APU_{i,0}, APW_{i,1}, \ldots, APW_{i,k})$ be a time series of the available CPU processing power of an agent a_i during different consecutive k measurement intervals. Let avg_i denote the average available CPU processing power of series S_i. We measure the stability value σ_i of agent a_i by calculating the standard deviation of the series S_i.

A small value of σ_i indicates that there has not been much change in the CPU utilization over previous periods of measurements. This value is used to predict how the utilization of the node is expected to be in the near future. However, the stability measure of the local node is not enough since any changes in the neighboring peers might trigger a reconfiguration. Therefore the node also senses how stable its surrounding environment is. The stability value is also carried in the WRM messages within the machine performance profiles. Therefore, every node records the stability values σ_j of its peers. The nodes periodically calculate σ, the standard deviation of the σ_j's of its peers.

A merging operation is triggered only when σ_i and σ are small, $\sigma_i < \epsilon_1$, and $\sigma < \epsilon_2$. In other words, the local node attempts to perform a merge operation

$$avg = \frac{1}{p} * \sum_{i=0}^{p} \sigma_i \tag{4.19}$$

$$\sigma = \sqrt{\frac{1}{p} * \sum_{i=0}^{p} (\sigma_i - avg)^2} \tag{4.20}$$

when possible if its surrounding environment is expected to be stable and the context switching rate of the hosting operating system is higher than a given threshold value. The OS context switching rates can be measured using tools such as the Unix `vmstat` command.

4.5 Related Work

Several research efforts have addressed the issue of load balancing in distributed heterogeneous environments at various system levels. Network-level load balancing tries to optimize the utilization of existing network resources by controlling traffic flow and minimizing the number of over-utilized links and under-utilized links [90]. Middleware-based load balancing provides the most flexibility in terms of balancing the load to different types of applications [82]. It is not constrained to the OS or network level but it spans different system levels.

Different load balancing strategies exist that range from static to dynamic, centralized to distributed, and sender-initiated to receiver-initiated strategies. Static approaches assume having *a priori* knowledge about the characteristics of the applications, the network, and the computing nodes. Static load balancing decisions are made at compile time. The static approach is simple and incurs no run-time overhead, however it is based on assumptions that are no longer valid in grid environments. First, it has to predict *a priori* the characteristics of applications. Second, grid environments are highly dynamic and their resource usage is constantly changing. Therefore, it is hard if not impossible to predict the characteristics of the execution environment statically. In contrast, dynamic load balancing strategies rely on collecting runtime resource usage information to make more informed decisions about how to balance the load across the nodes. Despite the complexity and overhead associated

with this approach, it has been widely argued that dynamic load balancing strategies outperform static approaches especially in dynamic environments. Dynamic load balancing strategies can be further categorized as centralized or decentralized. In the centralized scheme, resource information is collected and managed centrally by a single node. This node is responsible for collecting information about all the other nodes and properly redistributing the work. An obvious shortcoming of this strategy is scalability as the central node becomes a bottleneck as the system size increases and is highly vulnerable to failures. In a decentralized approach, every node attempts to make decisions by using local information or information collected from its neighbors. Most of the decentralized strategies use only partial information because of the high cost involved in obtaining global knowledge. Most of the existing approaches assume a homogeneous environment and/or ignore the underlying network topology of the nodes. The main goal of the proposed strategies is to design algorithms that account for the unique characteristics of grid environments, mainly the sheer size, heterogeneous resources, and wide range of latencies. We also propose strategies that are application communication topology sensitive and aware of the virtual interconnection of the underlying resources.

This work focuses on middleware-triggered decentralized load balancing strategies across large scale, highly dynamic peer-to-peer networks. Several load balancing strategies have been studied for structured and unstructured P2P systems. Some of them distribute objects across structured P2P systems [85, 96, 55]. They are all based on the concept of distributed hash tables. However they assume that all objects are homogeneous and have the same size. Rao et al. [62] have accounted for heterogeneity by using the concept of virtual servers that move from heavy nodes to light nodes which is similar in concept to migration of actors. However they assume that the load on virtual servers is stable. They also assume that there is only one bottleneck resource that needs to be optimized at a time.

Triantafillou et al. [107] have suggested load balancing algorithms to distribute contents over unstructured P2P systems. They aggregate global meta-data over a two-level hierarchy and they use it to re-assign objects. Our load balancing decision functions are not restricted to optimizing a specific bottleneck resource. Our mid-

dleware is not restricted to one load balancing strategy. It has been designed and implemented with the intention of plugging in different load balancing strategies depending on the nature of the running applications. This allows us to create concepts and decision functions that are application-specific, where placement and granularity of the application's entities can be modified dynamically.

4.6 Chapter Summary

We have described IOS reconfiguration mechanisms. The presented load balancing algorithms are based on decentralized work-stealing peer-to-peer protocols. Another key feature of the load balancing algorithms is their sensitivity to the network topology and the application's characteristics and their support for environments where resources join and leave dynamically. The virtual topology of IOS agents determines how resource information is exchanged between the peers and hence where reconfiguration is triggered. We also present the resource-sensitive model that determines the selection criteria of entities to be reconfigured. This model takes into account the application's characteristics, the remaining life expectancy of the application's entities, and how many times they have migrated in the past, to make more informed decisions about which entities to select for reconfiguration. The goal of this model is to co-locate tightly-coupled entities within the same node. The split policy complements the load balancing policies by providing the appropriate entities' granularity to achieve the desired load adjustment. On the other hand, the merge policy is a local policy that attempts to improve the node-level performance by reducing the context-switching overhead associated with having a large number of entities in the same node.

CHAPTER 5

Reconfiguring MPI Applications

To allow MPI applications to benefit from the reconfiguration policies embodied by the IOS middleware, we have developed a user-level library on top of MPI that allows process migration and malleability. Our strategy achieves portability across different implementations of the MPI standard. MPI/IOS is a system that integrates IOS middleware strategies with existing MPI applications. MPI/IOS adopts a semi-transparent checkpointing mechanism, where the user guides the reconfiguration mechanism by specifying the data structures that must be saved and restored to allow process migration. This approach does not require extensive code modifications. Legacy MPI applications can benefit from load balancing features by inserting just a small number of calls to a simple application programming interface.

This chapter starts by motivating the need for reconfigurable MPI in Section 5.1. Next we present the adopted approach to reconfigurable MPI applications in Section 5.2. The Process Checkpointing, Migration, and Malleability (PCM) library is discussed in Section 5.3. MPI/IOS runtime architecture is explained in Section 5.4. We then discuss the protocol used to interface between the PCM library and IOS in Section 5.5. Section 5.6 presents related work. The chapter concludes with discussion and summary in Section 5.7.

5.1 Motivation

While grid environments present multiple resource management challenges, they also provide an abundant pool of resources that is appealing to large scale and computationally demanding distributed applications. Examples are parallel computational science and engineering applications that arise in diverse disciplines such as astrophysics, fluid dynamics, materials science, biomechanics, or nuclear physics. These applications involve solving or simulating multi-scale problems with dynamic behavior. Solution procedures use sophisticated adaptive methods underlying data structures (e.g., meshes) and numerical methods to achieve specified levels of solution accuracy [32]. This adaptivity, when used for parallel solution procedures, introduces load imbalance which can be corrected using application-level dynamic load balancing techniques [102]. These applications generally deal with huge amounts of data and require extensive computational resources, but they usually assume a fixed number of cooperating processes running in a dedicated and mostly homogeneous computing environment or they address heterogeneity but not dynamic environments [103]. Running such applications on computational grids, with their dynamic, heterogeneous, and non-dedicated resources, makes it difficult for application-level load balancing alone to take full advantage of available resources and to maintain high performance. Application-level load-balancing approaches have a limited view of the external world where the application is competing for resources with several other applications. Middleware is a more appropriate location where to place resource management and load balancing capabilities since it has a more global view of the execution environment, and can benefit a large number of applications.

A large number of scientific and engineering applications have been implemented using the Message Passing Interface (MPI) [77] to achieve parallelism. MPI [77] has been widely adopted as the *de-facto* standard to implement single-program multiple-data (SPMD) parallel applications. Extensive development effort has produced many large software systems that use MPI for parallelization. Its wide availability has enabled portability of applications among a variety of parallel computing environments. However, the issues of scalability, adaptability and load balancing still remain a challenge. Most existing MPI implementations assume a static network environ-

ment. MPI implementations that support the MPI-2 standard [50, 78] provide partial support for dynamic process management, but still require complex application development from end-users: process management needs to be handled explicitly at the application level, which requires the developer to deal with issues such as resource discovery and allocation, profiling, scheduling, load balancing, etc. Additional middleware-support for application reconfiguration is therefore needed to relieve application developers from such concerns. Augmenting MPI applications with automated process migration capabilities is a necessary step to enable dynamic reconfiguration through load balancing of MPI processes among geographically distributed nodes.

Feitelson and Rudolph [42] classify parallel applications into four categories from a scheduling perspective: *rigid*, *moldable*, *evolving*, and *malleable*. Rigid applications require a fixed allocation of processors. Once the number of processors is determined, the application cannot run on a smaller or larger number of processors. Moldable applications can run on various numbers of processors. However, the allocation of processors remains fixed during the runtime of the application. In contrast, both evolving and malleable applications can change the number of processors during execution. In case of evolving applications, the change is triggered by the application itself. While in malleable applications, it is triggered by an external resource management system. We further extend the definition of malleability by allowing the parallel application not only to change the number of processors in which it runs but also to change the granularity of its processes.

5.2 Approach to Reconfigurable MPI Applications

5.2.1 The Computational Model

We use the SPMD model where each task executes the same program but operates on different data sets. We assume that applications are iterative and consist of one or more phases, which are sections of code comprised of computation followed by communication. Our model assumes that applications use a variable block distribution, where a contiguous set of data items are assigned to each node.

We target in this work the broad class of iterative applications. A large number

of scientific and engineering applications exhibit an iterative nature. The solution of such problems requires dividing up the physical domain into small elements or volumes to form what is referred to as a *mesh*. Such problems arise in various fields such as fluid dynamics, particle physics, and material science. Such problems are usually computationally intensive and cannot be solved by a single machine in a reasonable time. Parallelization techniques are then used to allow these problems to be solved in a distributed memory environment, where several processors are used simultaneously to achieve the desired solution. Parallelization consists primarily of partitioning the mesh into a number of subdomains. Each subdomain forms a separate process and is assigned to a single processor. We have chosen to experiment with the class of iterative applications for two reasons: 1) this class is important in the scientific and engineering communities, and 2) it exhibits predictable profiling and reconfiguration points that could easily be automated through static software analysis or code-level annotations.

Iterative applications are characterized by having a sequence of instructions that are repeated several times during the course of execution. The iterations continue until the problem converges or the error estimation is below a specified threshold. For example in a typical fluid dynamics solver, the iterations are used to solve the flow equations. Every iteration uses the existing pressure field to calculate a velocity field that satisfies momentum conservation. Then it tries to correct the pressures and velocities to satisfy continuity. The application keeps iterating until the solution converges. In this specific case the data domain is represented as a triangular or tetrahedral mesh.

Iterative mesh-based applications usually involve communication between their subdomains. In every iteration, the subdomains may need to synchronize by exchanging data. Applications could range from loosely synchronous to highly synchronous depending on how much synchronization is needed by the different subdomains. This feature forces the solution performance for the highly synchronized subdomains to depend on the performance of the slowest or most overloaded processor in the computation environment.

5.2.2 Process Migration

In MPI, any communication between processes needs to be done as part of a communicator. An MPI communicator is an opaque object with a number of attributes, together with simple functions that govern its creation, use and destruction. An intracommunicator delineates a communication domain which can be used for point-to-point communications as well as collective communication among the members of the domain. On the other hand, an intercommunicator allows communication between processes belonging to disjoint intracommunicators.

MPI processes periodically get notified by the middleware of migration or reconfiguration requests. When a process receives a migration notification, it initiates checkpointing of its local data in the next synchronization point. Checkpointing is achieved through library calls that are inserted by the programmer in specific places in the application code. The class of iterative applications exhibits natural locations (at the beginning of each iteration) to place polling, checkpointing and resumption calls. When the process is first started, it checks whether it is a fresh process or it has been migrated. In the second case, it proceeds to data and process interconnectivity restoration.

We achieve MPI process migration by rearranging MPI communicators. Migration is performed by a collaboration of all the participating MPI processes. It has to be done at a point where there are no pending communications. Process migration requires careful update of any communicator that involves the migrating process. A migration request forces all running MPI processes to enter a reconfiguration phase where they all cooperate to update their shared communicators. The migrating process spawns a new process in the target location and sends it its locally checkpointed data. Figure 5.1 describes the steps involved in managing the MPI communicators for a sample process migration. In the original communicator, Communicator 1, Process 7 has received a migration request. Process 7 cooperates with the processes of Communicator 1 to spawn the new process, Process 0 in Communicator 2, which will eventually replace it. The intercommunicator that results from this spawning is merged into one global communicator. Later, the migrating process is removed from the old communicator and the new process is assigned rank 7. The new process

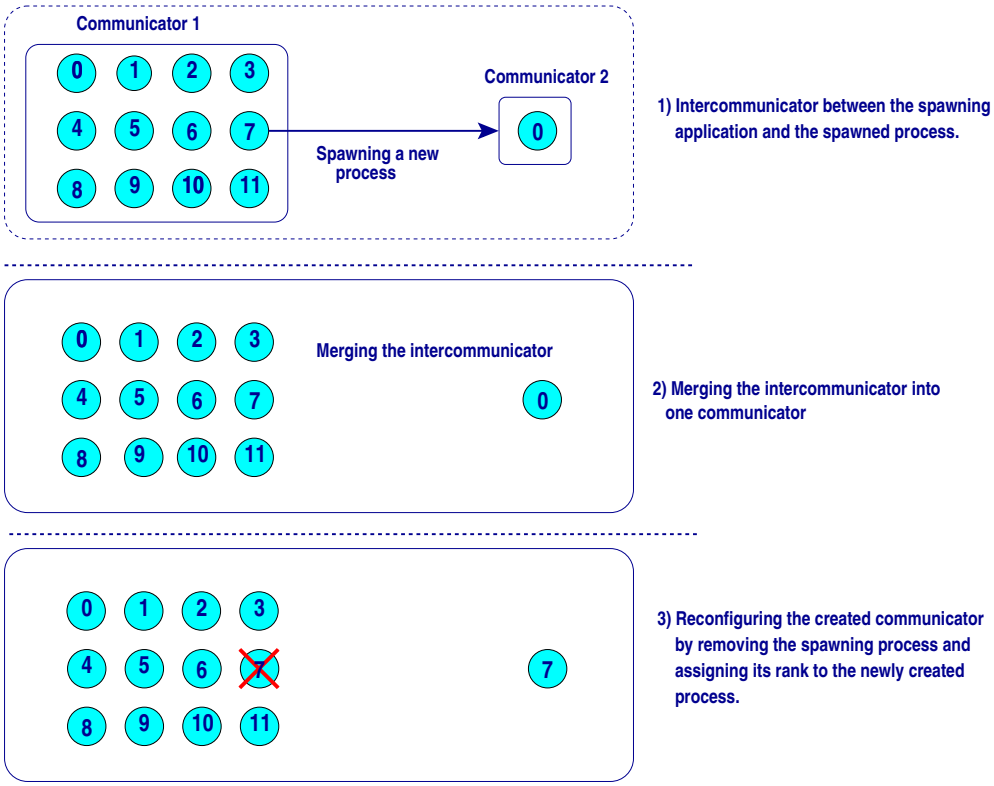

Figure 5.1: Steps involved in communicator handling to achieve MPI process migration.

restores the checkpointed data from its local daemon and regains the same state of the migrating process. All processes then get a handle to the new communicator and the application resumes its normal execution.

5.2.3 Process Malleability

We discussed in the previous section how we have implemented process migration in iterative MPI applications. However, load balancing using only process migration is limited by the application's process granularity over shared and dynamic environments [37]. In such environments, it is impossible to accurately predict the availability of resources at application's startup and hence determine the right granularity of the application. Hence, an effective alternative is to allow applications' processes to expand and shrink opportunistically as the availability of the resources changes dynamically. Over-estimating by starting with a very small granularity might

degrade the performance in case of a shortage of resources. At the same time, underestimating by starting with a large granularity might limit the application from potentially utilizing more resources. A better approach is therefore to enable dynamic process granularity changes through malleability.

There are operational and meta-level issues that need to be addressed when deciding how to reconfigure applications though malleability and/or migration. Operational issues involve determining how to split and merge the application's processes in ways that preserve the semantics and correctness of the application. The operational issues are heavily dependent on the application's programming model. On the other hand, meta-level issues involve deciding when should a process split or merge, how many processes to split or merge, and what is the proper mapping of the processes to the physical resources. These issues render programming for malleability and migration a complex task. To facilitate an application's reconfiguration from a developer's perspective, middleware technologies need to address meta-level reconfiguration issues. Similarly, libraries need to be developed to address the various operational issues at the application-level. This separation of concerns allows the meta-level reconfiguration policies built into middleware to be widely adopted by various applications.

Several design parameters come to play when deciding how to split and merge an application's parallel processes. Usually there is more than one process involved in the split or merge operations. The simplest scenario is performing binary split and merge, which allows a process to split into two processes or two processes to merge into one. Binary malleable operations are more intuitive since they mimic the biological phenomena of cell division. They are also highly concurrent since they could be implemented with a minimal involvement of the rest of the application. Another approach is to allow a process to split into N processes or N processes to merge into 1. This approach, in the case of communication intensive applications, could increase the communication overhead significantly and could limit the scalability of the application. It could also easily cause data imbalances. This approach would be useful when there are large fluctuations in resources. The most versatile approach is to allow for collective split and merge operations. In this case, the semantics of the split

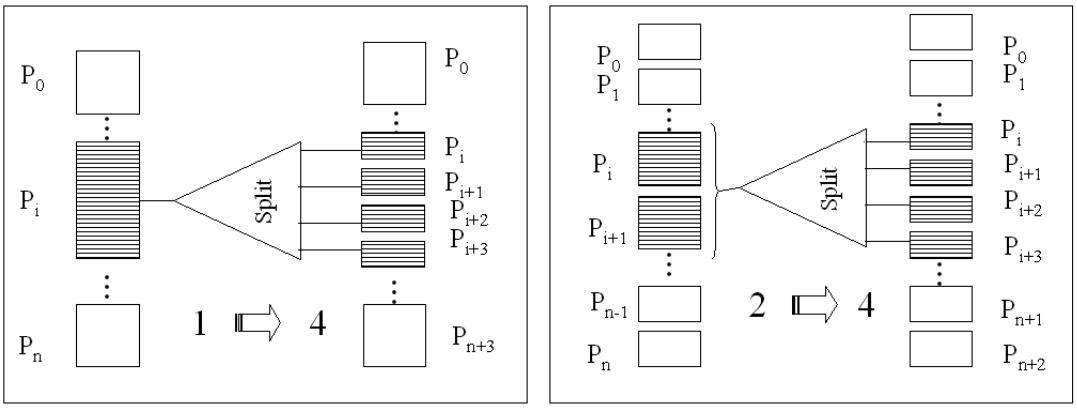

Figure 5.2: Example M to N split operations.

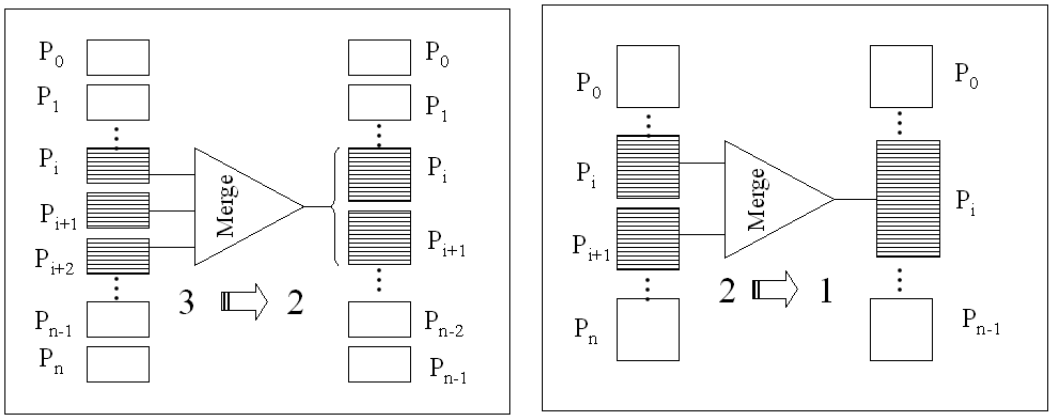

Figure 5.3: Example M to N merge operations.

or merge operations allow any number of M processes to split or merge into any other number of N processes. Figures 5.2 and 5.3 illustrate example behaviors of split and merge operations. In the case of the M to N approach, data is redistributed evenly among the resulting processes when splitting or merging. What type of operation is more useful depends on the nature of applications, the degree of heterogeneity of the resources, and how frequently the load fluctuates.

While process migration changes mapping of an application's processes to physical resources, split and merge operations go beyond that by changing the communication topology of the application, the data distribution, and the data locality. Splitting and merging causes the communication topology of the processes to be modified be-

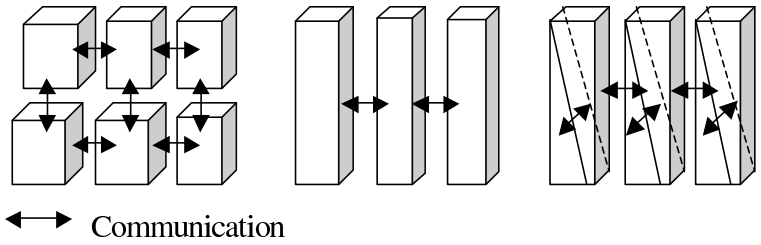

Figure 5.4: Examples of domain decomposition strategies showing block, column, and diagonal decompositions for a 3D data-parallel problem.

cause of the addition of new or removal of old processes, and the data redistribution among them. This reconfiguration needs to be done atomically to preserve application semantics and data consistency.

We provide high-level operations for malleability based on the MPI paradigm for SPMD data parallel programs with regular communication patterns. The proposed approach is high level in that the programmer is not required to specify when to perform split and merge operations and some of the intrinsic details involved in re-arranging the communication structures explicitly: these are provided by the PCM library. The programmer needs, however, to specify the data structures that will be involved in the malleability operations. Since there are different ways of subdividing data among processes, programmers also need to guide the split and merge operations for data-redistribution.

5.3 The Process Checkpointing Migration and Malleability Library

To support migration and malleability in iterative MPI programs, we have designed and implemented an application-level checkpointing API called Process Checkpointing, Migration, and Malleability (PCM) and a runtime system called Process Checkpointing, Migration, and Malleability Daemon (PCMD). Few PCM calls need to be inserted in MPI programs to specify the data that need to be checkpointed, to

restore the process to its previous state in case of migration or malleability, to update the data distribution structure and the MPI communicator handles, and to probe the runtime system for reconfiguration requests. This library is semi-transparent because the user does not have to worry about when or how checkpointing and restoration is done. The underlying PCMD infrastructure takes care of all the checkpointing, migration, and malleability details.

PCM is implemented entirely in the user-space for portability of the checkpointing, migration, and malleability schemes across different platforms. PCM has been implemented on top of MPICH2 [61], a freely available implementation of the MPI-2 standard.

5.3.1 The PCM API

PCM provides four classes of services: environmental inquiry services, checkpointing services, global initialization and finalization services, and collective reconfiguration services. Table 5.1 shows the classification of the PCM API calls. `PCM_MPI_Init` is a wrapper for `MPI_Init`. The user calls this function at the beginning of the program. `PCM_MPI_Init` is a collective operation that takes care of initializing several internal data structures. It also reads a configuration file that has information about the port number and location of the PCM daemon, a runtime system that provides checkpointing and global synchronization between all running processes.

Migration and malleability operations require the ability to save and restore the current state of the process(es) to be reconfigured. `PCM_Store` and `PCM_Load` provide storage and restoration services of the local data. Checkpointing is handled by the PCMD runtime system that ensures that data is stored in locations with reasonable proximity to their destination.

Upon startup, an MPI process can have three different states:

1. `PCM_STARTED`: a process that has been initially started in the system (for example, using `mpiexec`),

2. `PCM_MIGRATED`: a process that has been spawned because of a migration, and

Service Type	Function Name
Initialization	PCM_MPI_Init
Finalization	PCM_Exit, PCM_Finalize
Environmental Inquiry	PCM_Process_Status PCM_Comm_rank PCM_Status PCM_Merge_datacnts
Reconfiguration	PCM_Reconfigure PCM_Migrate PCM_Split, PCM_Merge PCM_Split_Collective PCM_Merge_Collective
Checkpointing	PCM_Load, PCM_Store

Table 5.1: The PCM API.

3. `PCM_SPLITTED`: a process that has been spawned because of a split operation.

A process that has been created as a result of a reconfiguration (migration or split) proceeds to restoring its state by calling `PCM_Load`. This function takes as parameters information about the keys, pointers, and data types of the data structures to be restored. An example includes the size of the data, the data buffer and the current iteration number. Process ranks may also be subject to changes in case of malleability operations. `PCM_Comm_rank` reports to the calling process its current rank. Conditional statements are used in the MPI program to check for its startup status. An illustration is given in Figure 5.6, which shows the instrumentation in the initialization phase and Figure 5.7, which shows the instrumentation in the iteration phase.

The running application probes the PCMD system to check if a process or a group of processes need to be reconfigured. Middleware notifications set global flags in the PCMD system. To prevent every process from probing the runtime system, the root process (usually process with rank 0) probes the runtime system and broadcasts any reconfiguration notifications to the other processes. This provides a callback mechanism that makes probing non-intrusive for the application. `PCM_Status` returns the state of the reconfiguration to the calling process. It returns different values to different processes. In the case of a migration, `PCM_MIGRATE` value is returned to the process that needs to be migrated, while `PCM_RECONFIGURE` is returned to the

other processes. `PCM_Reconfigure` is a collective function that needs to be called by both the migrating and non-migrating processes. Similarly `PCM_SPLIT` or `PCM_MERGE` are returned by the `PCM_Status` function call in case of a split or merge operation. All processes collectively call the `PCM_Split` or `PCM_Merge` functions to perform a malleable reconfiguration.

We have implemented the 1 to N and M to N split and merge operations. `PCM_Split` and `PCM_Merge` provide the 1 to N behavior, while `PCM_Split_Collective` and `PCM_Merge_Collective` provide the M to N behavior. The middleware is notified about which form of malleability operation to use implicitly. The values of M and N are transparent to the programmer. They are provided by the middleware which decides the granularity of the split operation.

Split and merge functions change the ranking of the processes, the total number of processes, and the MPI communicators. All occurrences of `MPI_COMM_WORLD`, the global communicator with all the running processes, should be replaced with `PCM_COMM_WORLD`[4]. This latter is a malleable communicator since it expands and shrinks as processes get added or removed. All reconfiguration operations happen at synchronization barrier points. The current implementation requires no communication messages to be outstanding while a reconfiguration function is called. Hence, all calls to the reconfiguration PCM calls need to happen either at the beginning or end of the loop.

When a process or group of processes engage in a split operation, they determine the new data redistribution and checkpoint the data to be sent to the new processes. When the new processes are created, they inquire about their new ranks and load their data chunks from the PCMD. The checkpointing system maintains an up-to-date database of data checkpoints per process rank. Then all of an application's processes synchronize to update their ranks and their communicators. The malleable calls return handles to the new ranks and the updated communicator. Unlike a split operation, a merge operation entails removing processes from the MPI communicator. Merging operations for data redistribution are implemented using MPI scatter and gather operations.

[4]The PCM library (v. 0.2) does not handle user-defined communicators. All processes must communicate using the `MPI_COMM_WORLD` global communicator.

5.3.2 Instrumenting an MPI Program with PCM

Figure 5.5 shows a sample skeleton of an MPI-based application with a very common structure in iterative applications. The code starts by performing various initializations of some data structures. Data is distributed by the root process to all other processes in a block distribution. The `xDim` and `yDim` variables denote the dimensions of the data buffer. The program then enters the iterative phase where processes perform computations locally and then exchange border information with their neighbors. Figures 5.6 and 5.7 show the same application instrumented with PCM calls to allow for migration and malleability. In case of split and merge operations, the dimensions of the data buffer for each process might change. The PCM split and merge take as parameters references to the data buffer and dimensions and update them appropriately. In case of a merge operation, the size of the buffer needs to be known so enough memory can be allocated. The `PCM_Merge_datacnts` function is used to retrieve the new buffer size. This call is significant only at processes that are involved in a merge operation. Therefore a conditional statement is used to check whether the calling process is merging or not. The variable `merge_rank` will have a valid process rank in the case the calling process is merging, otherwise it has the value -1.

The example shows how to instrument MPI iterative applications with PCM calls. The programmer is required only to know the right data structures that are needed for malleability. A PCM-instrumented MPI application becomes malleable and ready to be reconfigured by IOS middleware.

5.4 The Runtime Architecture

MPI/IOS is implemented as a set of middleware services that interact with running applications through an MPI wrapper.

The MPI/IOS runtime architecture consists of the following components (see Figure 5.8):

1. the PCM-enabled MPI applications,

2. the wrapped MPI that includes the PCM API, the PCM library, and wrappers

```
#include <mpi.h>
...

int main(int argc, char **argv) {
    //Declarations
    ....

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPLCOMM_WORLD, &rank );
    MPI_Comm_size( MPLCOMM_WORLD, &totalProcessors );

    current_iteration = 0;

    //Determine the number of columns for each processor.
    xDim = (yDim-2) / totalProcessors;

    //Initialize and Distribute data among processors
    ...

    for(iterations=current_iteration; iterations<TOTAL_ITERATIONS;
        iterations++){

        // Data Computation.
        ...

        //Exchange of computed data with neighboring processes.
        // MPI_Send() || MPI_Recv()
        ...
    }

    // Data Collection
    ...
    MPI_Barrier( MPLCOMM_WORLD );

    MPI_Finalize();
    return 0;
}
```

Figure 5.5: Skeleton of the original MPI code of an MPI application.

```
#include "mpi.h"
#include "pcm_api.h"
...

MPI_Comm PCM_COMM_WORLD;
int main(int argc, char **argv) {
    //Declarations
    ....
    int current_iteration, process_status;
    PCM_Status pcm_status;

    //declarations for malleability
    double *new_buffer;
    int merge_rank, mergecnts;

    PCM_MPI_Init(&argc, &argv);
    PCM_COMM_WORLD = MPI_COMM_WORLD;
    PCM_Init(PCM_COMM_WORLD);
    MPI_Comm_rank(PCM_COMM_WORLD, &rank );
    MPI_Comm_size(PCM_COMM_WORLD, &totalProcessors );
    process_status = PCM_Process_Status();

    if(process_status == PCM_STARTED){
        current_iteration = 0;

        //Determine the number of columns for each processor.
        xDim = (yDim-2) / totalProcessors;

        //Initialize and Distribute data among processors
        ...
    }
    else{
        PCM_Comm_rank(PCM_COMM_WORLD, &rank);
        PCM_Load(rank, "iterator",&current_iteration);
        PCM_Load(rank, "datawidth", &xDim);
        prevData = (double *)calloc((xDim+2)*yDim, sizeof(double));
        PCM_Load(rank, "myArray",prevData);
    }
            ...
            ...
}
```

Figure 5.6: Skeleton of the malleable MPI code with PCM calls: initialization phase.

```
...
   for(iterations=current_iteration; iterations<TOTAL_ITERATIONS;
       iterations++){
     pcm_status = PCM_Status(PCM_COMM_WORLD);
     if(pcm_status == PCM_MIGRATE){
       PCM_Store(rank,"iterator",&iterations,PCM_INT,1);
       PCM_Store(rank,"datawidth",&xDim,PCM_INT,1);
       PCM_Store(rank,"myArray",prevData,PCM_DOUBLE,(xDim+2)*yDim);
      PCM_COMM_WORLD = PCM_Reconfigure(PCM_COMM_WORLD, argv[0]);
     }
     else if(pcm_status == PCM_RECONFIGURE){
       PCM_Reconfigure(&PCM_COMM_WORLD, argv[0]);
       MPI_Comm_rank(PCM_COMM_WORLD, &rank);
     }
     else if(pcm_status == PCM_SPLIT){
       PCM_Split(prevData,PCM_DOUBLE,
                 &iterations,&xDim,&yDim,&rank,
                 &totalProcessors,&PCM_COMM_WORLD, argv[0]);
     }else if(pcm_status == PCM_MERGE){
       PCM_Merge_datacnts(xDim,yDim,&mergecnts,&merge_rank,
                          PCM_COMM_WORLD);
       if(rank == merge_rank)
         /*Reallocate memory for the data buffer*/
         new_buffer = (double*)calloc(mergecnts, sizeof(double));

       PCM_Merge(prevData,MPI_DOUBLE,&xDim,&yDim,new_buffer,
                 mergecnts,&rank,&totalProcessors,&PCM_COMM_WORLD);
       if(rank == merge_rank)
           prevData = new_buffer;
     }
     // Data Computation.
       ...
     //Exchange of computed data with neighboring processes.
     // MPI_Send() || MPI_Recv()
       ...
   }
   // Data Collection
   ...
   MPI_Barrier( PCM_COMM_WORLD );
   PCM_Finalize(PCM_COMM_WORLD);
   MPI_Finalize();
   return 0;
}
```

Figure 5.7: Skeleton of the malleable MPI code with PCM calls: iteration phase.

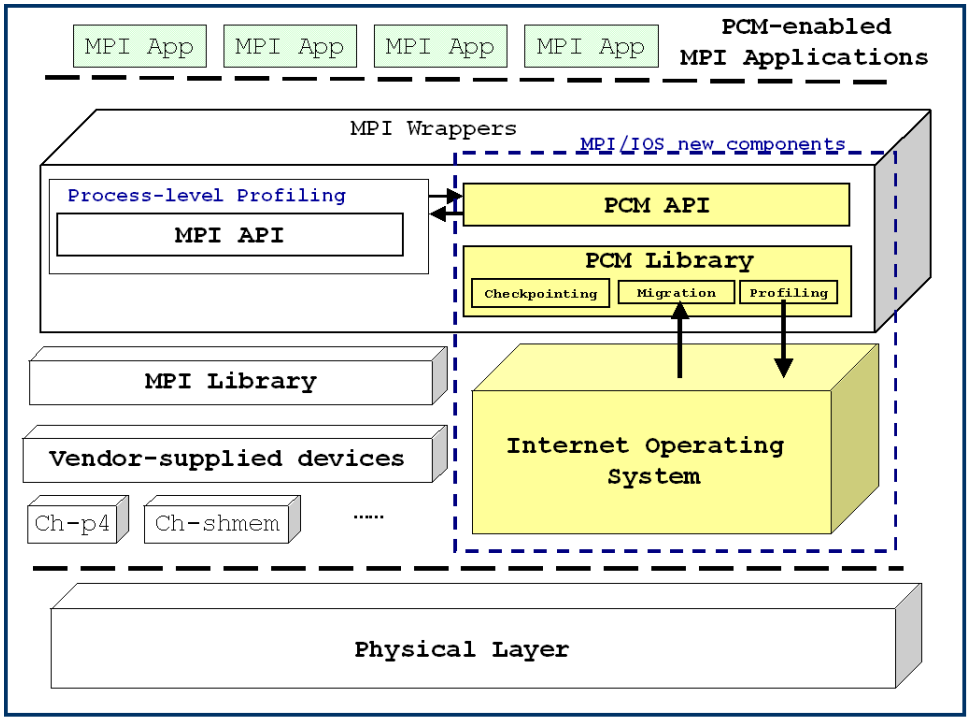

Figure 5.8: The layered design of MPI/IOS which includes the MPI wrapper, the PCM runtime layer, and the IOS runtime layer.

for all MPI native calls,

3. the MPI library, and

4. the IOS runtime components.

5.4.1 The PCMD Runtime System

Figure 5.9 shows an MPI/IOS computational node running MPI processes. A PCM daemon (PCMD) interacts with the IOS middleware and MPI applications. A PCMD is started in every node that actively participates in an application. A PCM dispatcher is used to start PCMDs in various nodes and used to discover existing ones. The application initially registers all MPI processes with their local daemons. The port number of a daemon is read from a configuration file that resides in the same host.

Every PCMD has a corresponding IOS agent. There can be more than one MPI

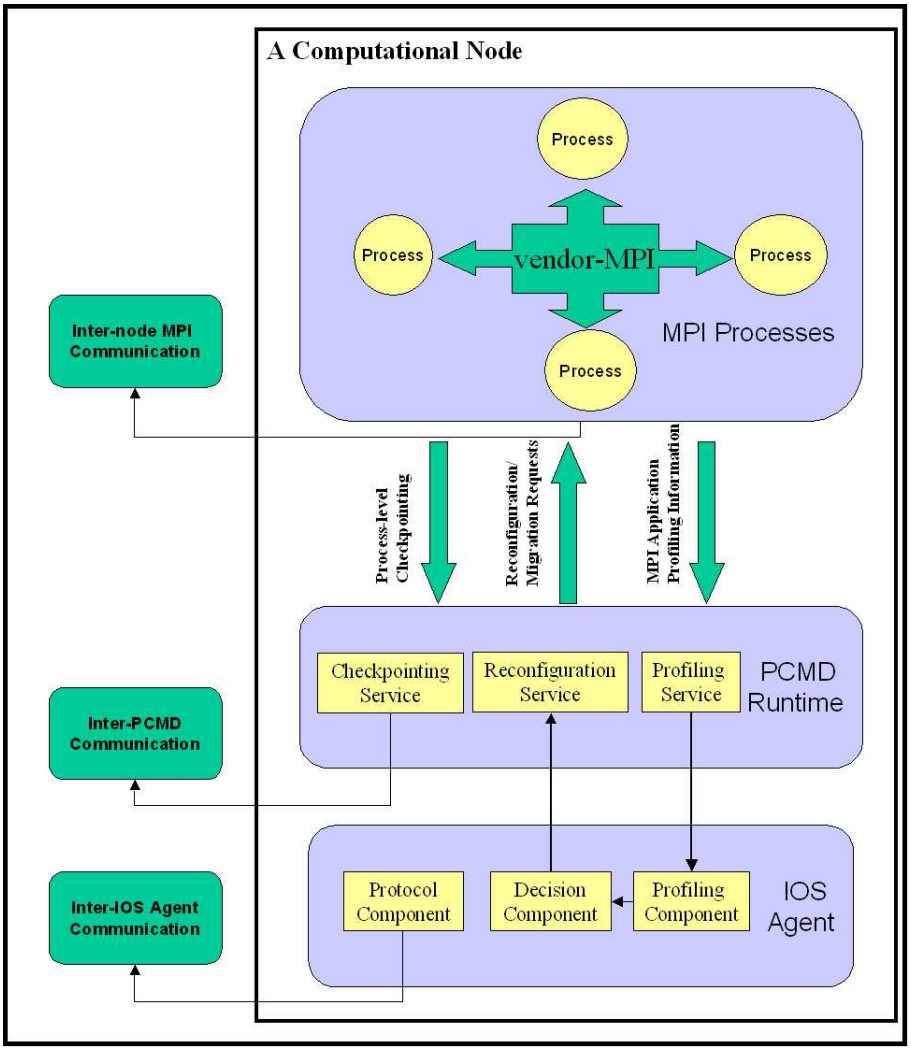

Figure 5.9: Architecture of a node running MPI/IOS enabled applications.

process in each node. The daemon consists of various services used to achieve process communication profiling, checkpointing, migration, and malleability. The MPI wrapper calls record information pertaining to how many messages have been sent and received and their source and target process ranks. The profiled communication information is passed to the IOS profiling component. IOS agents keep monitoring their underlying resources and exchanging information about their respective loads.

When a node's available resources fall below a predefined threshold or a new idle node joins the computation, a Work-Stealing Request Message (WRM) message is propagated among the actively running nodes. The IOS agent of a node responds

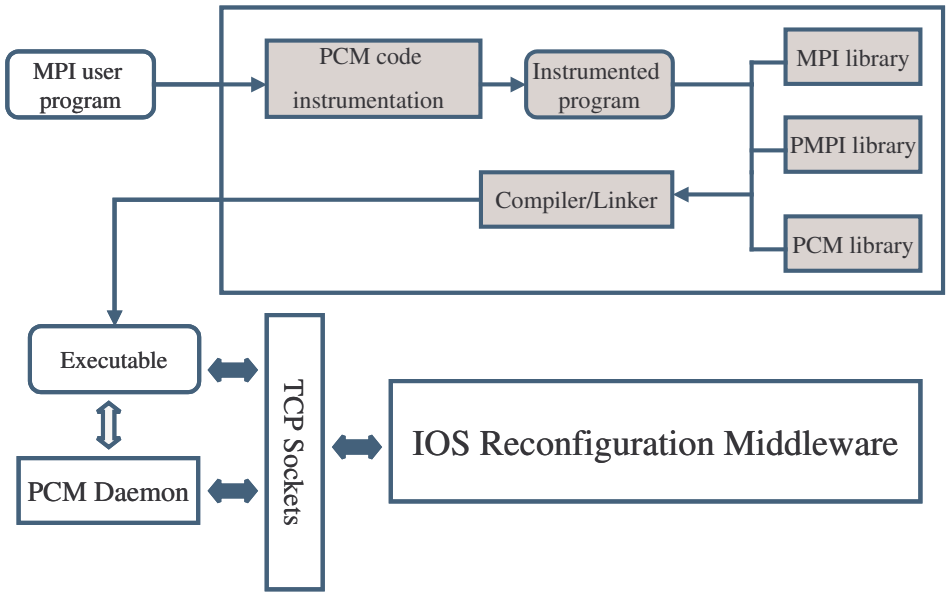

Figure 5.10: Library and executable structure of an MPI/IOS application.

to work-stealing requests if it becomes overloaded and its decision component decides according to the resource sensitive model which process(es) need(s) to be migrated. Otherwise, it forwards the request to an IOS agent in its set of peers. The decision component then notifies the reconfiguration service in the PCMD, which then sends a migration, split, or merge request to the desired process(es). At this point, all active PCMDs in the system are notified about the event of a reconfiguration. This causes all processes to cooperate in the next iteration until migration is completed and application communicators have been properly updated. Although this mechanism imposes some synchronization delay, it ensures that no messages are being exchanged while process migration is taking place and avoids incorrect behaviors of MPI communicators.

5.4.2 The Profiling Architecture

MPI processes need to send periodically their communication patterns to their corresponding IOS profiling agents. To achieve this, we have built a profiling library that is based on the MPI profiling interface (PMPI). The MPI specification provides a general mechanism for intercepting calls to MPI functions using name shifting. This

allows the development of portable performance analyzers and other tools without access to the MPI implementation source code. The only requirement is that every MPI function be callable by an alternate name (`PMPI_Xxxx` instead of the usual `MPI_Xxxx`.). The built profiling library intercepts all communication methods of MPI and sends any communication event to the profiling agent.

All profiled MPI routines call their corresponding `PMPI_Xxxx` and, if necessary, PCM routines. Figure 5.10 shows the library structure of the MPI/IOS programs. The instrumented code is linked with the profiling library PMPI, the PCM library, and a vendor MPI implementation's library. The generated executable passes all profiled information to the IOS run-time system and also communicates with the local PCMD. This latter is responsible for storing local checkpoints and passing reconfiguration decisions across a socket API from the IOS agent to the MPI processes. Section 5.5 describes in details the interaction protocol used between the PCM library and IOS runtime system.

5.4.3 A Simple Scenario for Adaptation

MPI applications interact with each other, with the checkpointing and migration services provided by the PCM library, and with the profiling and reconfiguration services provided by IOS agents. Walking through the simple scenario of an application adaptation that is shown in Figure 5.11 further explains these interactions. The example illustrates the execution of a parallel MPI application that is composed of n processes running on n different processors.

1. An available processor joins the IOS virtual network.
2. The new processor starts requesting work from its peers.
3. Processor 1 receives the work-stealing request. The decision component in its local IOS agent predicts that there will be a future performance gain migrating process 1 to the remote processor f.
4. MPI process 1 gets notified of a migration event.
5. At the beginning of the next iteration, the migrating process broadcasts a message to the rest of the processors so that they enter a reconfiguration phase.

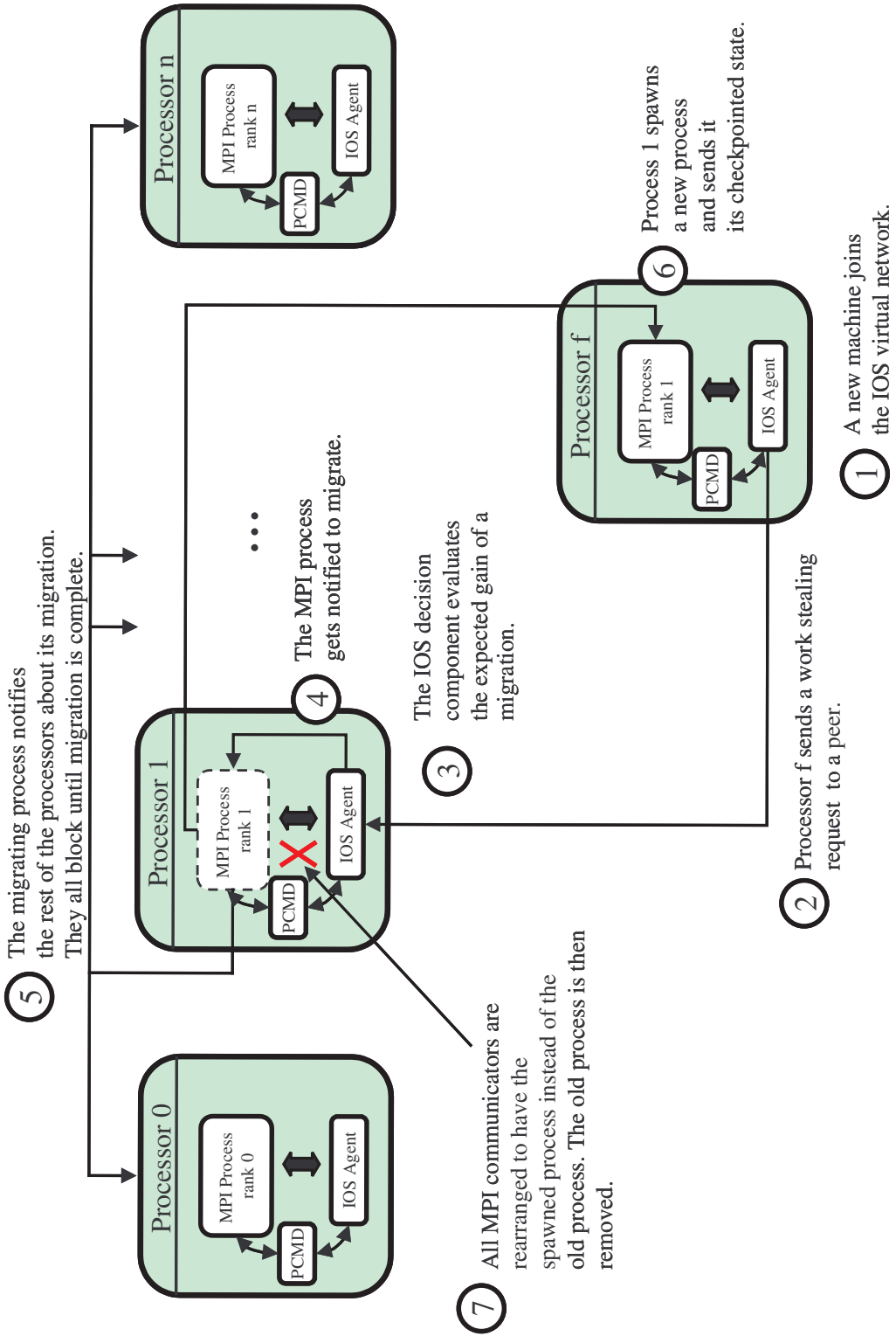

Figure 5.11: A reconfiguration scenario of an MPI/IOS application.

6. The migrating process checkpoints its local state, and spawns an image of itself in the remote processor f. The local PCMD takes care of transferring the checkpointed state to the newly created process and notifying the rest of the processes.

7. As a part of completing the migration process, the PCM library takes care of rearranging the `MPI_COM_WORLD` communicator by removing the old process and including the new one. The newly created process gets assigned rank 1.

5.5 The Middleware Interaction Protocol

The IOS/MPI proxy is a software server that runs on all nodes wishing to interact with the IOS middleware. The purpose of the proxy is to enable the communication between reconfigurable MPI processes and IOS agents. The proxy also implements an actor emulation mechanism whereby MPI processes are viewed by the IOS middleware as universal actors. This allows MPI processes to take advantage of the autonomous nature of actors such as universal naming, asynchronous message passing, and migration capabilities. This approach also allows the IOS middleware to be independent from the high-level programming model and not to require any specific changes to reconfigure MPI applications.

5.5.1 Actor Emulation

When a process is initialized, it must be registered with the proxy. Process registration involves creating an *ActorEmulator* object at the level of the proxy to represent the running process. The process is assigned a UAN/UAL pair following SALSA's universal naming mechanisms (see Table 5.5.1 for an example of the naming scheme adopted). The process also registers with the naming server, which keeps track of process IDs and their current locations. The same procedure happens anytime a new process get created as a result of a split operation or an existing process migrates to another host. It registers with the local proxy, which updates the naming server with the new UAN/UAL mapping. The actor emulation mechanism allows MPI processes to leverage SALSA's migration and naming protocols. These protocols

UAN	uan://<name server: port>/<process name>/<process rank>
UAL	rmsp://<local host:port>/<process name>/<process rank>
Example	uan://io.wcl.cs.rpi.edu:3030/heat/1 rmsp://europa.wcl.cs.rpi.edu:4040/heat/1

Table 5.2: The structure of the assigned UAN/UAL pair for MPI processes at the MPI/IOS proxy.

ensure that the name server has an up-to-date mapping between process IDs and their locations.

5.5.2 The Proxy Architecture

The proxy consists of two channel servers: an MPI channel server and an IOS channel server. The MPI channel server is responsible for getting the profiling messages from an MPI process and forwarding them to the IOS profiling module. This server is also responsible for translating MPI-specific profiling messages into IOS-specific profiling messages. The MPI channel server interfaces with the profiling module through the IOS profiling interface. The IOS channel server acts as a theater that hosts the ActorEmulator objects. It maintains the list of all processes running in the local host and intercepts any reconfiguration requests targeted to them. The IOS channel server sends any intercepted reconfiguration messages (migrate, split, or merge) to the targeted MPI process through the PCM daemon.

Figure 5.12 shows the flow of information between a reconfigurable MPI process, the proxy, and the IOS agent. MPI processes send profiling messages to the IOS profiling module through the MPI channel server using a text-oriented TCP socket-based protocol. The protocol module sends reconfiguration requests to the MPI processes through the IOS channel server. The low level communication between the IOS channel server and the protocol actor is done using the RMSP protocol.

5.5.3 Protocol Messages

Two types of messages are exchanged between MPI processes and the proxy. Control messages carry control-related information that pertains to reconfiguration requests and process registration. Table 5.5.3 shows the supported types of control

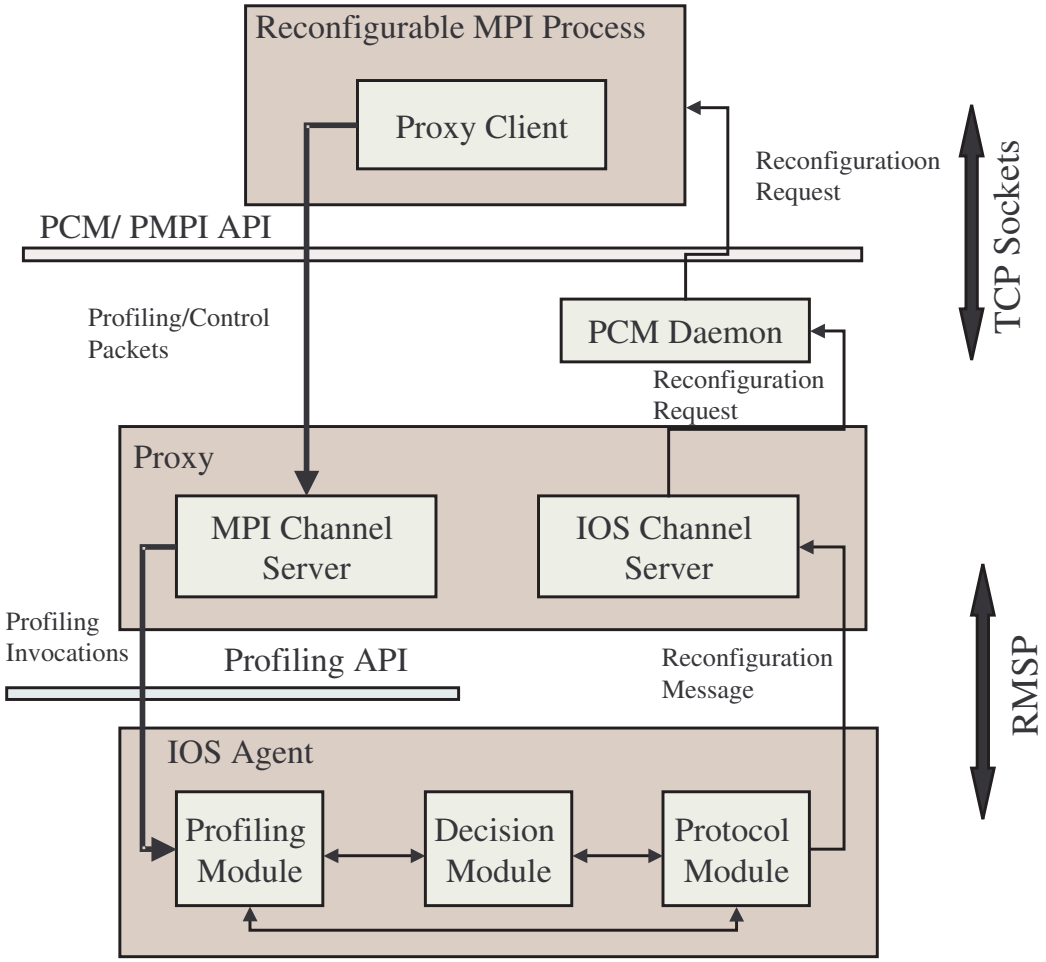

Figure 5.12: IOS/MPI proxy software architecture.

messages and their codes. Profiling messages carry profiled information about process communication events and some application-specific metrics such as iteration-time. Table 5.5.3 shows the supported types of profiling messages. The proxy protocol can easily be extended to support more types.

Figure 5.13 shows the structure of the protocol messages. Both messages have the same header structure which carries the version of the protocol, the size of the message in bytes, and the type of the message (control or profiling). The `SRCID` and `DESTID` fields in the profiling message indicate the `ID`s of the source and destination processes for the event that is profiled. For example if an `MPI_Send` message is profiled, these IDs will correspond to the source and destination ranks. The `COMM TYPE` field

Control Message Type	Code in Hexadecimal
CONNECT	0x0000
PROFILE	0x0001
MIGRATE	0x0002
DISCONNECT	0x0003
ENDPROCESSED	0x0004
ENDMIGRATE	0x0005

Table 5.3: Proxy control packets types.

Profiling Message Type	Code in Hexadecimal
SEND	0x00
RECV	0x01
BCAST	0x02
GATHER	0x03
SCATTER	0x04

Table 5.4: Proxy profiling message types.

refers to the type of the communication event being profiled (send, receive, scatter, gather, etc). The `SIZE` field is the size of the message being sent in the profiled event. In the control message, the `SRCID` refers to the rank of the process to whom the control message is targeted. The interpretation of the IP address depends on the type of the message. If this is a `migrate` message, the IP refers to the remote host where the process needs to migrate. Otherwise it refers to the local IP. The split and merge control messages have different formats. Split and merge messages are another type of control messages. The `MAL` field carries the degree of the malleable operation. In other words, it is the number that the processes need to split to or the number they need to merge to. The `SIZE` field is the number of processes that will engage in the split or merge operation. Then `SIZE` number of `ID`s follow in the message that indicate the ranks of the processes that need to be engage in the malleable operation.

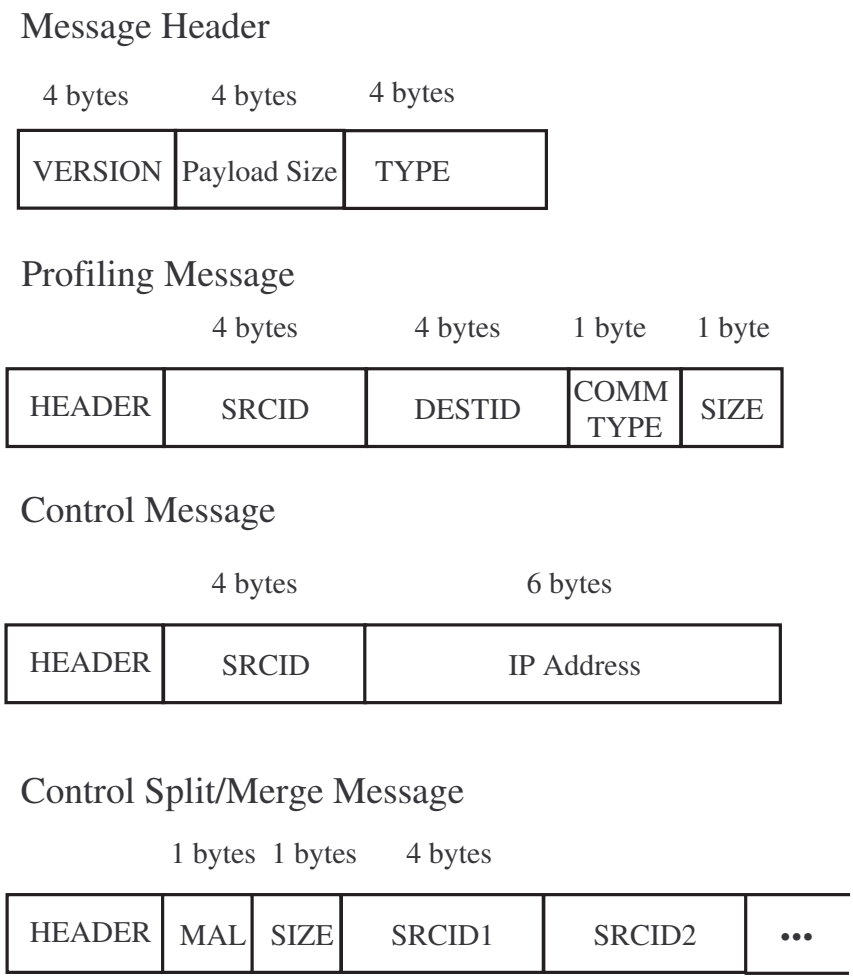

Figure 5.13: The packet format of MPI/IOS proxy control and profiling messages.

5.6 Related Work

5.6.1 MPI Reconfiguration

There are a number of conditions that can introduce computational load imbalances during the lifetime of an application:

1. the application may have irregular or unpredictable workloads from, e.g., adaptive refinement,

2. the execution environment may be shared among multiple users and applications,

3. the execution environment may be heterogeneous, providing a wide range of processor speeds, network bandwidth and latencies, and memory capacity, and/or

4. the execution environment may be dynamic where nodes join and leave.

Dynamic load balancing (DLB) is necessary to achieve a good parallel performance when such imbalances occur. Most DLB research has targeted the application level (e.g., [38, 44]), where the application itself continuously measures and detects load imbalances and tries to correct them by redistributing the data, or changing the granularity of the problem through domain repartitioning. Although such approaches have proved beneficial, they suffer from several limitations. First, they are not transparent to application programmers. They require complex programming and are domain specific. Second, when these applications are run on a dynamic grid, application-level techniques which have been applied successfully to heterogeneous clusters [38, 103] may fall short in coping with the high fluctuations in resource availability and usage. Our research targets middleware-level DLB which allows a separation of concerns: load balancing and resource management are transparently dealt with by the middleware, while application programmers deal with higher level domain specific issues.

Several recent efforts have focused on enhancing MPI run-time systems to adapt applications to dynamic environments. Adaptive MPI (AMPI) [19, 57] is an implementation of MPI on top of light-weight threads that balances the load transparently based on a parallel object-oriented language with object migration support. AMPI leverages Charm++ dynamic load balancing. Load balancing in AMPI is done through migrating user-level threads that MPI processes are executed on. This approach limits the portability of process migration across different architectures since it relies on thread migration. Process swapping [93] is an enhancement to MPI that uses over-allocation of resources and improves performance of MPI applications by allowing them to execute on the best performing nodes. MPI process swapping has been also used for the class of iterative applications. Our approach is different in the sense that we do not need to over-allocate resources initially. Such a strategy, though potentially very useful, may be impractical in large-scale dynamic environments such as grids where resources join and leave and where an initial over-allocation may not be possible. We allow new nodes that become available to join the computational

grid to improve the performance of running applications during their execution.

Other efforts have focused on process checkpointing and restart as a mechanism to allow applications to adapt to changing environments. Examples include CoCheck [95], starFish [6], MPICH-V [22], and the SRS library [110]. CoCheck, starFish, and MPICH-V support checkpointing for fault-tolerance, while we provide this feature to allow process migration and hence load balancing. Our framework could be integrated with the sophisticated checkpointing techniques used in these projects to be able to support also non-iterative applications. SRS supports checkpointing to allow application stop and restart. Our work differs in the sense that we support migration at a finer granularity. Application-transparent process checkpointing is not a trivial task, could be very expensive, and is architecture-dependent as it requires saving the entire application state. Semi-transparent checkpointing provides a simpler solution and a more portable approach. It has been proved useful for the important class of iterative applications [93, 110]. API calls are inserted in the MPI program that informs the middleware of the important data structures to save. This is an attractive solution that can benefit a wide range of applications and does not incur significant overhead since only relevant state is saved. The instrumentation of applications could be easily automated since iterative applications have a common structure.

5.6.2 Malleability

Existing approaches to application malleability have focused on processor virtualization (e.g [58]) by allowing the number of processes in a parallel application to be much larger than the number of available processors. This strategy allows flexible and efficient load balancing through process migration. Processor virtualization can be beneficial as more and more resources join the system. However, when resources slow down or become unavailable, certain nodes can end up with a large number of processes. The node-level performance is then impacted by the large number of processes it hosts because the small granularity of each process causes unnecessary context-switching overhead and increases inter-process communication. On the other hand, having a large process granularity does not always yield the best performance

because of the memory-hierarchy. In such cases, it is more efficient to have processes with data that can fit in the lower level of memory caches' hierarchy.

Malleability for MPI applications has been mainly addressed through processor virtualization, dynamic load balancing strategies, and application stop and restart.

Malleability is achieved in AMPI [58] by starting the applications with a very fine process granularity and relying on dynamic load balancing to change the mapping of processes to physical resources through object migration. The PCM/IOS library and middleware support provide both migration and process granularity control for MPI applications. Phoenix [101] is another programming model which allows virtualization for a dynamic environment by creating extra initial processes and using a virtual name space and process migration to balance load and scale applications.

In [108], the authors propose virtual malleability for message passing parallel jobs. They apply a processor allocation strategy called the *Folding by JobType* (FJT) that allows MPI jobs to adapt to load changes. The folding technique reduces the partition size in half, duplicating the number of processes per processor. In contrast to our work, the MPI jobs are only simulated to be malleable by using moldability and the folding technique.

Dyn-MPI [116] is another system that extends iterative MPI programs with adaptive execution features in non-dedicated environments through data redistribution and the possibility of removing badly performing nodes. In contrast to our scheme, Dyn-MPI does not support the dynamic addition of new processes. In addition Dyn-MPI relies on a centralized approach to determine load imbalances, while we utilize decentralized load balancing policies to trigger malleable adaptation.

Checkpointing and application stop and restart strategies have been investigated as malleability tools in dynamic environments (e.g, CoCheck [95], starFish [6], and SRS [110]). Stop and restart is expensive especially for applications operating on large data sets. The SRS library [110] provides tools to allow an MPI program to stop and restart where it left off with a different process granularity. Our approach is different in the sense that we do not need to stop the entire application to allow for change of granularity.

5.7 Summary and Discussion

We described in this chapter the PCM library framework for enabling iterative MPI applications to be dynamically reconfigurable through migrate, split, and merge operations. The implementation of malleability operations is described and illustrated through an example of a communication-intensive iterative application. Different techniques for split and merge are presented and discussed. We also presented the PCM API and how it is used to instrument existing MPI programs. We have also shown how the PCM library has been integrated with the IOS middleware to allow for automatic reconfiguration capabilities. The integration relies on a simple socket-based protocol that is not restricted to MPI applications. In fact, any programming model or language that can speak the same protocol can communicate with the IOS middleware to send profiled information and get notifications about reconfiguration requests. The proxy architecture provides a bridging and translation mechanism between application-specific formats and IOS-specific formats. The actor emulation mechanism is also powerful in that it provides a virtualization mechanism that hides many of the IOS details from applications. The integration protocols emphasize a separation of concerns between high-level application programming models and IOS middleware.

We have investigated programming methodologies that promote a separation of the concerns in the implementation of computationally-intensive scientific applications on a large network of computers. High-level programming abstractions provide a natural interface to scientists so that they can concentrate on their domain of expertise. Programming tools map these high-level abstractions into executable units that support efficient communication, dynamic partitioning, and load balancing. Run-time middleware infrastructure supports adaptability of executing systems to an evolving underlying network. The presented programming paradigm, languages, and tools are a first step in the unification of parallel and distributed computing by enabling parallel systems to adapt to different and evolving distributed execution environments.

It is impractical to expect scientists to rewrite or even make significant modifications to extensive libraries of Fortran, C, and C++ software that currently use MPI. The MPI/IOS architecture allows application programs to run using native C/C++

code, continuing to use MPI for interprocess communication. Iterative applications can immediately take advantage of MPI/IOS functionality for dynamic resource allocation, process migration, and malleability.

CHAPTER 6

Performance Evaluation

This chapter discusses some key experimental results pertaining to the IOS middleware. We show the effect of using knowledge about the application's topology in the reconfiguration decisions and the effect of certain virtual topologies on the quality of reconfiguration. We also evaluate the effect of reconfiguration using both SALSA and MPI applications.

6.1 Experimental Testbed

For the experimental testbed we used a heterogeneous environments consisting of different clusters with various platforms, processing capabilities, and different network latencies and bandwidths:

- **Cluster A** consists of 4 dual-processor SUN Blade 1000 machines with a processing speed of 750MHz and 2 GB of memory. The SUN machines are interconnected with high-speed Gigabit Ethernet.

- **Cluster B** consists of 18 single-processor SUN Ultra 10 machines with a processing speed of 360MHz and 256 MB of memory. The machines are connected with 100 MB Ethernet.

- **Cluster C** consists of four quad-processor AIX single-core Power-PC processors running at 1.7GHz, with 64KB L1 cache, 6MB L2 cache and 128MB L3 cache. Each machine has 8GB RAM, for a total of 32GB RAM in the entire clus-

Figure 6.1: The two-dimensional heat diffusion problem.

ter. Intra-cluster communication is with 1GB/sec bandwidth, 100sec latency Ethernet.

- **Cluster D** is single-core Opteron cluster that consists of twelve quad-processor, single-core Opterons running at 2.2GHz, with 64KB L1 cache and 1MB L2 cache. Each machine has 16GB RAM, for a total of 192GB RAM. The machines are connected by 10GB/sec bandwidth, 7usec latency Infiniband.

- **Cluster E** is a dual-core Opteron cluster that consists of four quad-processor, dual-core Opterons running at 2.2GHz, with 64KB L1 cache and 1MB L2 cache. Each machine has 32GB RAM, for a total of 128GB RAM. The machines are connected with 1GB/sec bandwidth, 100sec latency Ethernet.

The AIX cluster (cluster C) is connected to the Opteron clusters (clusters D and E) over the RPI campus area network.

We also used various machines from the RPI computer science public labs: the SPARC laborarory which contains 20 Dell Optiplex GX300s running FreeBSD and the SunRay which contains 33 Sun Microsystems SunRay1 terminals.

6.2 Applications Case Studies

6.2.1 Heat Diffusion Problem

We have used a fluid dynamics application that solves heat diffusion in a solid for testing and validation purposes (see Figure 6.1). This application is representative of highly synchronized iterative mesh-based applications. We have implemented this

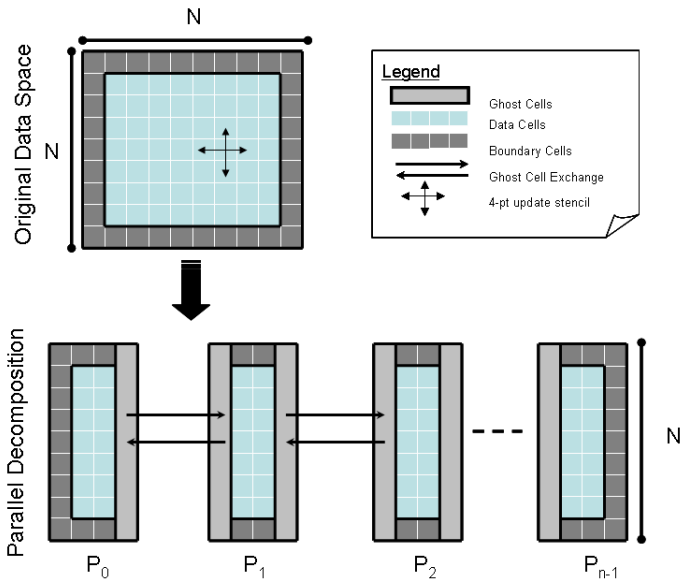

Figure 6.2: Parallel decomposition of the 2D heat diffusion problem.

application using C and MPI. The application's code has been instrumented with profiling points and reconfiguration points to allow gathering dynamic information about its performance progress and performing reconfiguration through migration.

Heat diffusion is a popular application that models the heat transfer in a solid. We have used a simplified version of this problem to evaluate our reconfiguration strategies. A two-dimensional mesh of cells is used to represent the problem data space. The mesh initially contains the initial values of the mesh with the boundary values. The cells are uniformly distributed among the parallel processors. At the beginning, a master process takes care of distributing the data among processors. For each iteration, the value of each cell is calculated with the values of its neighbor cells. Each cell needs to maintain a current version of the values of its neighboring cells. To achieve this, processors exchange values of the neighboring cells, also referred to as ghost cells. To sum up, every iteration consists of doing some computation and exchange of ghost cells from the neighboring processors. Figure 6.2 shows the structure of the parallel decomposition of the heat diffusion problem.

6.2.2 Search for the Galactic Structure

The search for the galactic structure application is part of the Sloan Digital Sky Survey (SDSS) project [4]. SDSS is a digital imaging and spectroscopic data bank of the celestial sphere. It covers most of the sky above galactic latitude of 30 degrees. All the images are taken from a big telescope located in New Mexico. SDSS data is being used to map out where the stars are in the Milky Way galaxy and to determine the structure of the galaxy. This will help in improving our understanding of how the galaxy formed in the first place. One of the most challenging tasks of mapping the stars of the galaxy is figuring out their distance from earth. The color of the stars is used to determine their brightness. The intrinsic brightness of the stars is used to estimate how far they are and hence to build a 3-dimensional shape of the galaxy.

The search of galactic structure application performs a maximum likelihood fit. Given some data and a model, it tries to find the probabilities that the model fits the data. This application is both data and computation intensive. This application can be categorized as a loosely coupled farmer/worker application. For each iteration, the farmer generates a model and the workers determine the accuracy of this model, calculated by using the model on each worker's individual set of star positions. The farmer then combines these results to determine the overall accuracy, determines modifications for the model and the process repeats. Each iteration involves testing multiple models to determine which model parameters to change, using large data sets (hundreds of thousands or millions of stars), resulting in a low communication to computation ratio, allowing for massive distribution.

6.2.3 SALSA Benchmarks

SALSA benchmarks were developed to model various communication topologies with an emphasis on the communication-to-computation ratio of the applications. The applications have been designed with one unit of computation per one unit of communication. Therefore, the more neighbors an actor has the higher the communication-to-computation ratio. Four different application topologies are represented, each pertaining to a level of inter-actor communication and representing different connectivity levels:

- **The unconnected topology**. A representative for massively parallel applications where actors continuously perform computations without exchanging any messages.
- **The sparse topology**. A model of applications that have a moderate level of communication.
- **The tree topology**. Actors are inter-connected in a tree structure to model a much higher degree of inter-actor communication than the sparse topology.
- **The hypercube topology**. This benchmark provides the highest amount of inter-actor communication modeling a very high communication to computation ratio.

6.3 Middleware Evaluation

6.3.1 Application-sensitive Reconfiguration Results

We ran the four SALSA benchmarks each pertaining to level of inter-actor communication on cluster A. The unconnected actor graph had actors simply process messages over and over, with no inter-actor communication. The sparse actor graph linked actors randomly, providing a moderate amount of inter-actor communication. The tree topology linked actors in a tree structure, for a higher amount of inter-actor communication. Lastly, the hypercube topology provided a very high amount of inter-actor communication. (see Figures 6.3, 6.4, 6.5 and 6.6). We compared throughput of RS and ARS to manual load balancing to measure the overhead that the IOS middleware incurred on the computation. All actors were placed in a round robin fashion across the eight theaters, then were allowed to compute until their throughput leveled off. Throughput is the number of messages processed by all actors in a given amount of time – the higher the throughput, the faster a computation is running.

Figure 6.3 shows that both ARS and RS imposed a minimal amount of overhead, as a round robin placement of actors is the optimal load balancing solution for an unconnected graph of actors in a homogeneous network, and the round robin placement imposed no middleware overhead. ARS and RS performed comparatively to RR in this test. On the more communication-bound topologies (see Figures 6.4, 6.5

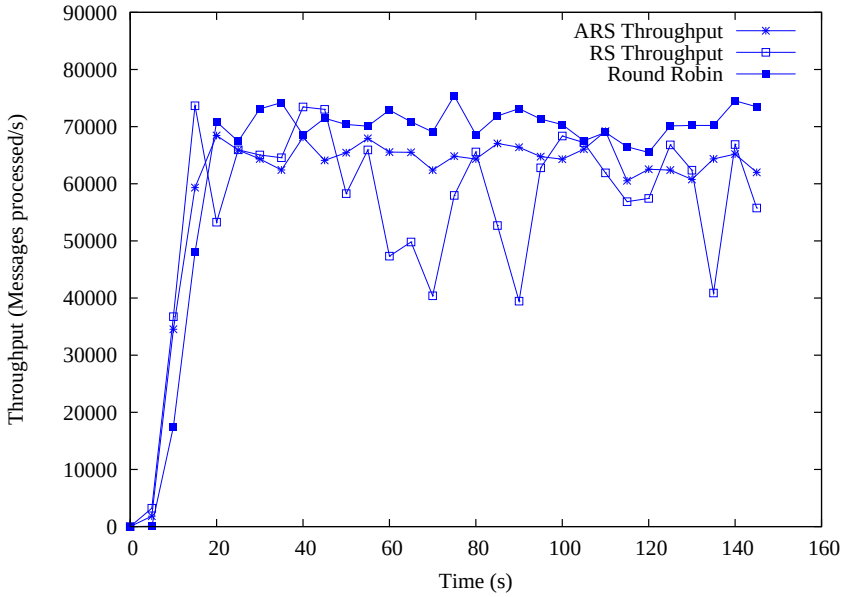

Figure 6.3: Performance of the massively parallel unconnected benchmark.

and 6.6), ARS outperformed both the manual load balancing and RS. On a sparsely connected graph, ARS performed superbly, bringing throughput to nearly the level of an unconnected graph. In all benchmarks involving inter-actor communication, ARS highly outperformed RR and RS, showing that the co-location of actors significantly improves message throughput. RS was shown to be too unstable and performed poorly compared to the RS and ARS strategies. This was mainly due to the fact that a random strategy failed to collocate actors correctly and resulted in a lot of thrashing behavior. Therefore actors spent most of their time migrating instead of doing useful work, which impacted negatively the overall application's throughput.

6.3.2 Experiments with Dynamic Networks

To show how IOS can handle a dynamically changing network, the same benchmarks were run on a dynamic network with joining and leaving nodes. The actors were placed initially on one node, then every 30 seconds an additional peer node was added to the computation. After having eight nodes join the computation, the applications were run for two minutes to balance the load, then one node was removed gradually every 30 seconds.

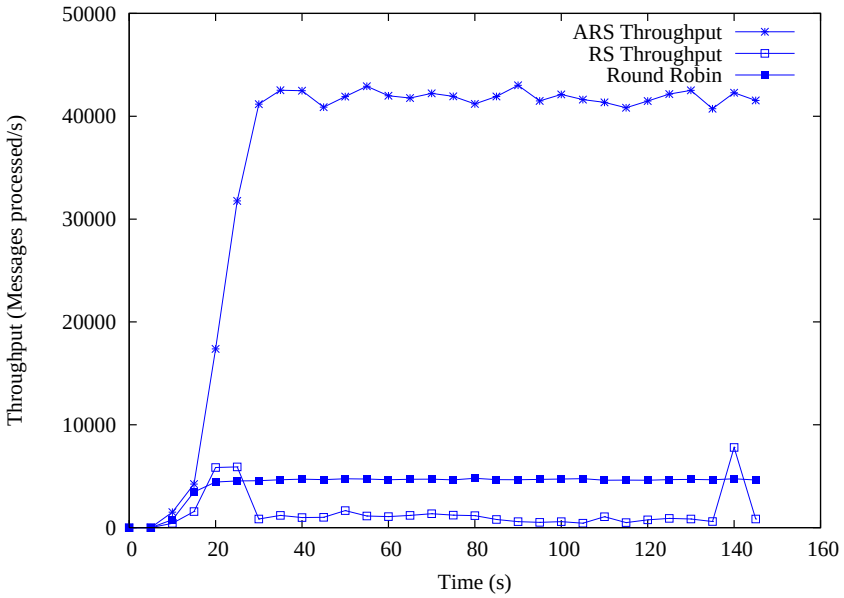

Figure 6.4: Performance of the massively parallel sparse benchmark.

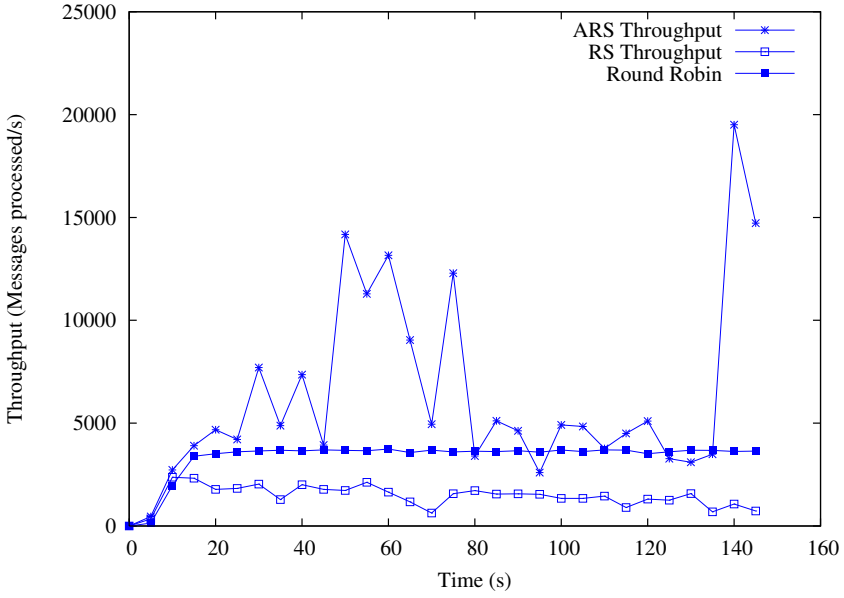

Figure 6.5: Performance of the highly synchronized tree benchmark.

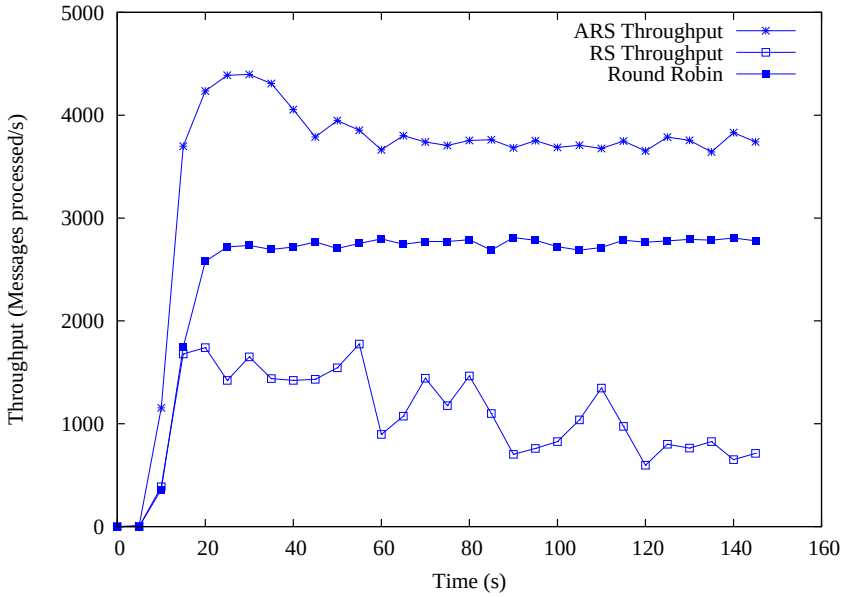

Figure 6.6: Performance of the highly synchronized hypercube benchmark.

With the unconnected graph join/leave topology (see Figure 6.8), both RS and ARS performed well in distributing the load across the peer nodes (see Figure 6.8), and increased the throughput by a factor of about six when all eight nodes had joined the computation. The addition and removal of peer theaters shows that IOS can rebalance load with removal and addition of nodes without much overhead.

In the tree join/leave scenario (see Figure 6.7) ARS performed well, increasing throughput by a factor of about 3.5 when all eight nodes were joined. RS however performed worse than simply placing the application entirely onto one node. The graphs of actor placement (see Figure 6.7) show that while both ARS and RS managed to distribute the actors evenly across the network of theaters, ARS co-located actors more appropriately according to their connectivity, significantly improving overall throughput.

These results show that the IOS system with ARS performs well in most situations for load balancing of an actor system. While the more traditional strategy of random stealing does not fare so well in an autonomous system of actors, a more intelligent strategy can exploit the properties of the actor model to provide autonomic solutions for load balancing across a dynamic network. The results also show that

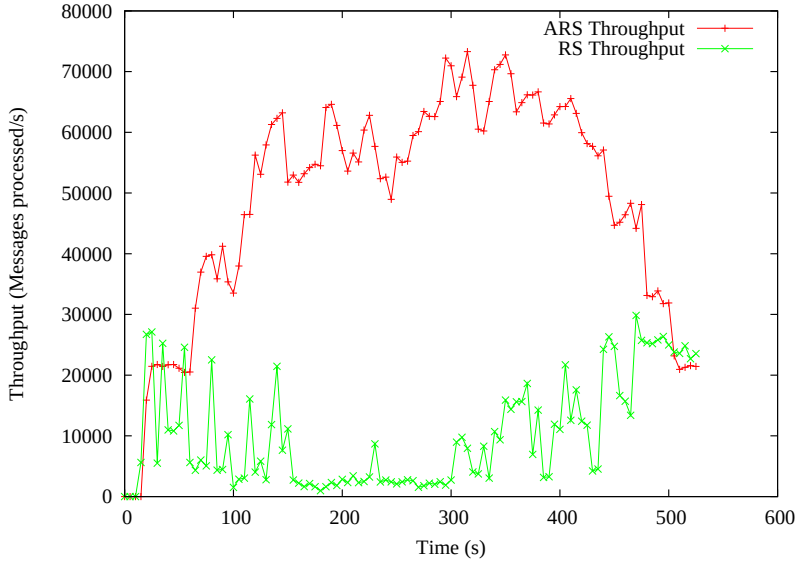

Figure 6.7: The tree topology on a dynamic environment using ARS and RS.

IOS can handle the addition and removal of nodes from a computation without any central coordination, a necessity for large dynamic heterogeneous networks.

6.3.3 Experiments with Virtual Topologies

We describe in this section the tests used in the evaluation of the cluster-to-cluster (NSc2c) and the peer-to-peer (NSp2p) virtual networks for load balancing, and the results of this evaluation.

Two different physical environments to model Internet-like networks and Grid-like networks were used to evaluate the effect of the virtual network topology on the load balancing decisions. The first physical network consists of 20 machines running Solaris and Windows operating systems with different processing power and different latencies to model the heterogeneity of Internet computing environments. The second physical network consists of 5 clusters with different inter-cluster network latencies. Each cluster consists of 5 homogeneous SUN Solaris machines. Machines in different clusters have different processing power.

For most application topologies, NSc2c performed better than NSp2p on grid-like environments (see Figures 6.10, 6.12, and 6.14). The results show the central coordination and knowledge of NSc2c allows more accurate reconfiguration with less

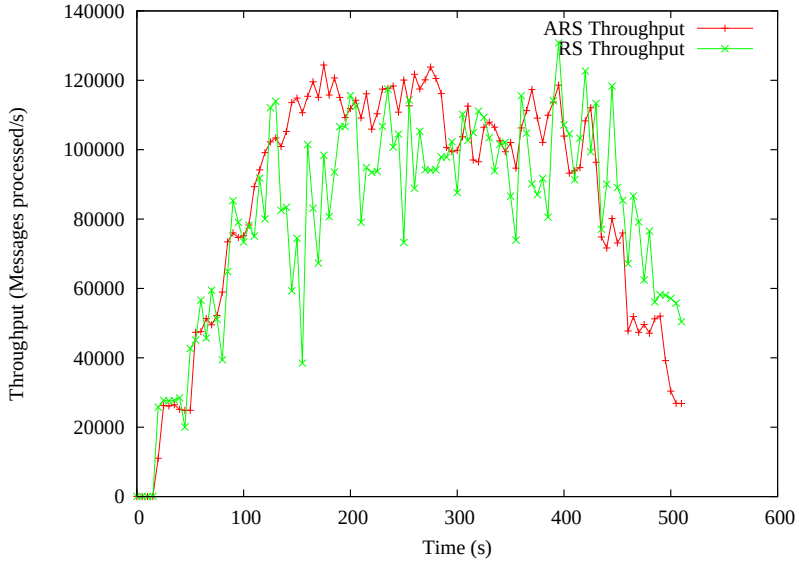

Figure 6.8: The unconnected topology on a dynamic environment using ARS and RS.

overhead due to the homogeneous nature of the clusters in the physical layer. NSp2p lacks this central coordination which explains the decreased performance.

In Internet-like environments, NSp2p outperformed NSc2c (see Figures 6.9, 6.11, and 6.15). Due to the lack of homogeneous resources which could be centrally coordinated, NSc2c required additional overhead and performed less accurate reconfiguration. NSp2p did not require the extra overhead of central management and thus proved to be a better strategy on this type of physical network.

However, in both cases NSp2p outperformed NSc2c in the unconnected application topology, as NSc2c outperformed NSp2p in the sparse application topology. This is due to the way the strategies interact with the application topologies. For the unconnected application topology, simply dispersing the actors as much as possible across the network achieves the best load balancing; however with the sparse application topology, more tightly coupled actors should be kept closer together. NSp2p more quickly disperses actors across the physical network, while NSc2c tries to keep actors within the same cluster only migrating actors out when necessary.

These experiments show that decentralized middleware management can accomplish intelligent application reconfiguration, but also that the virtual network used plays a role in the effectiveness of the reconfiguration. The p2p topology performs

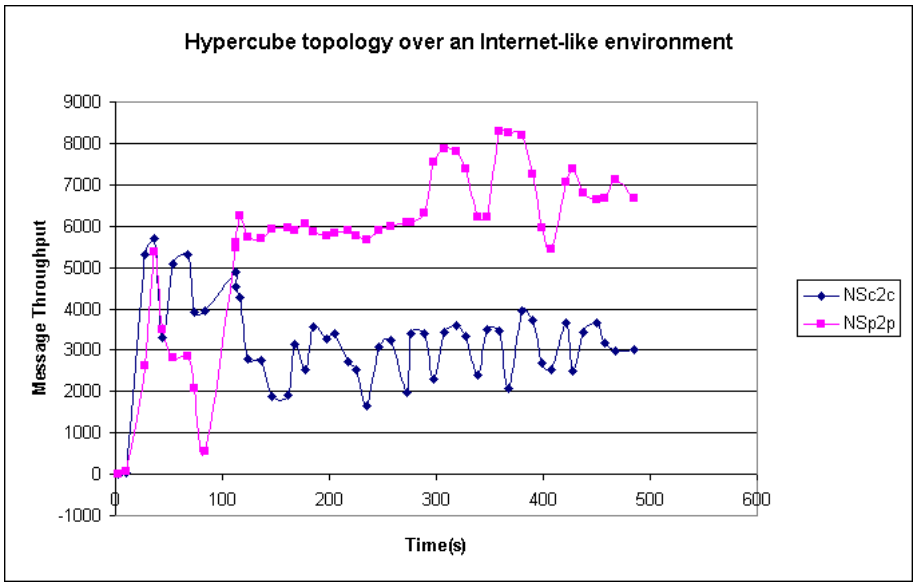

Figure 6.9: The hypercube application topology on Internet-like environments.

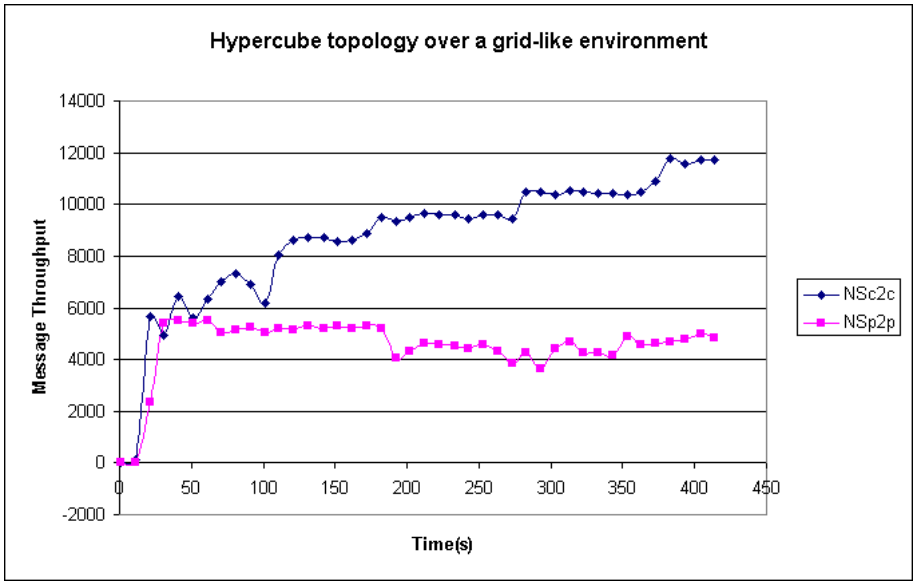

Figure 6.10: The hypercube application topology on Grid-like environments.

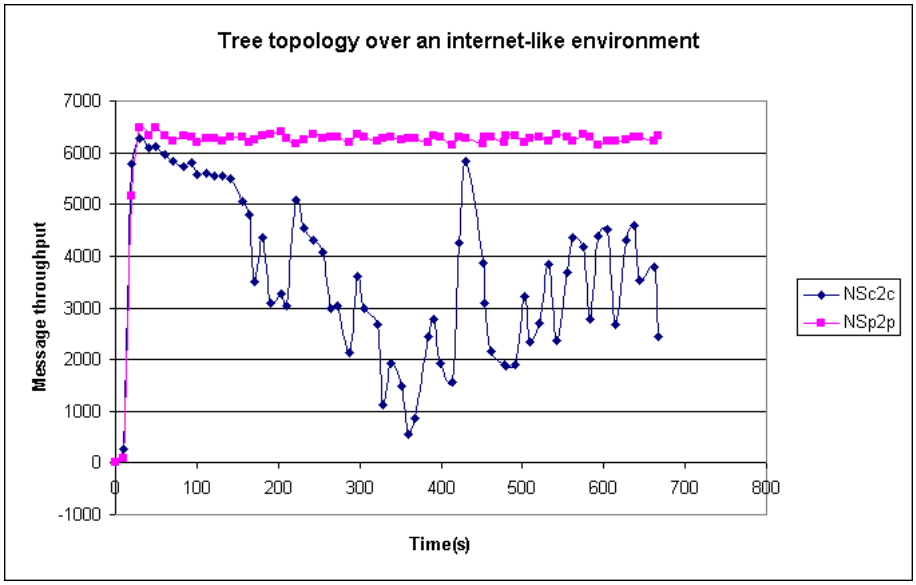

Figure 6.11: The tree application topology on Internet-like environments.

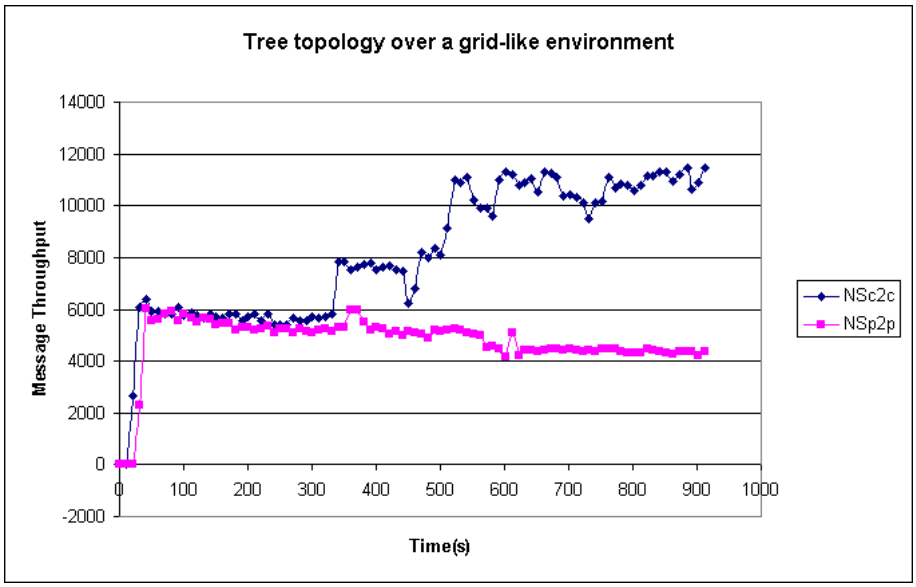

Figure 6.12: The tree application topology on Grid-like environments.

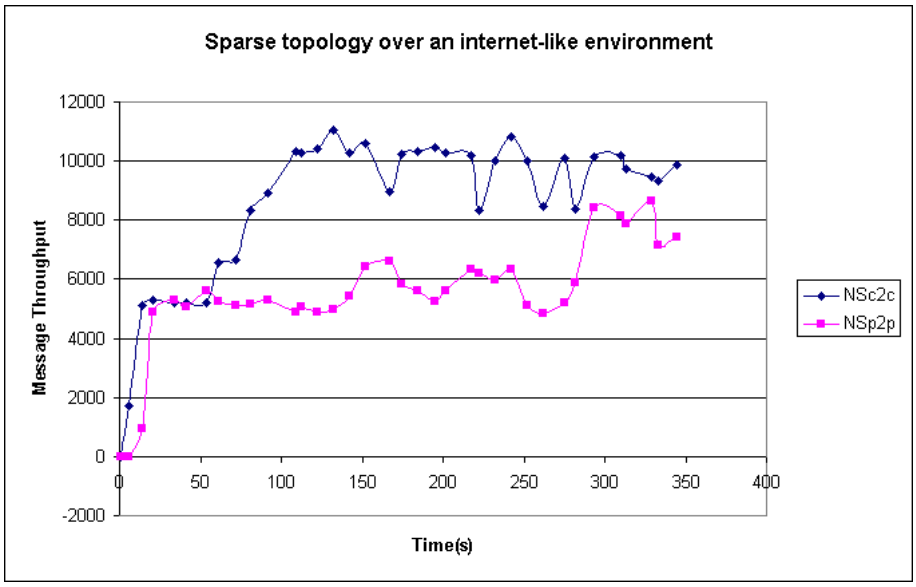

Figure 6.13: The sparse application topology on Internet-like environments.

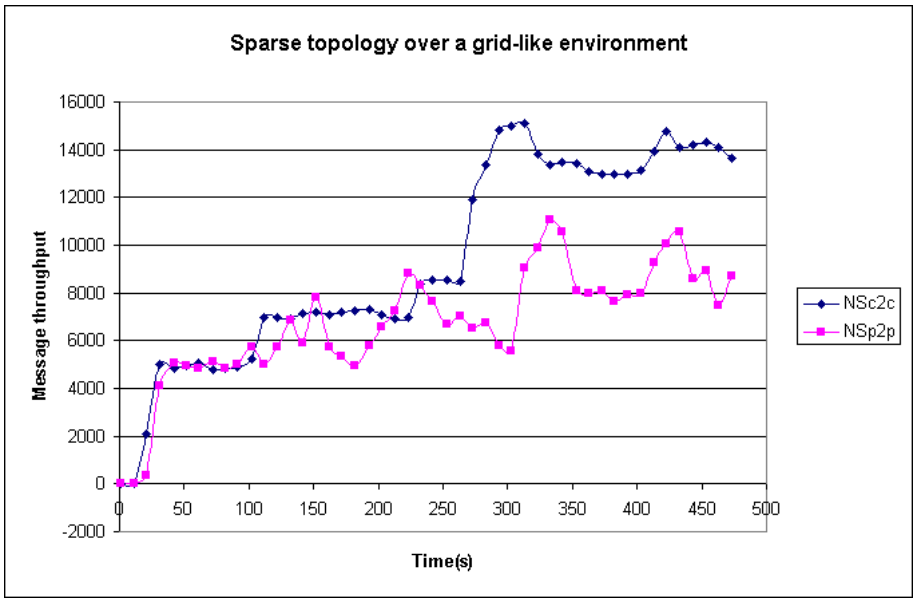

Figure 6.14: The sparse application topology on Grid-like environments.

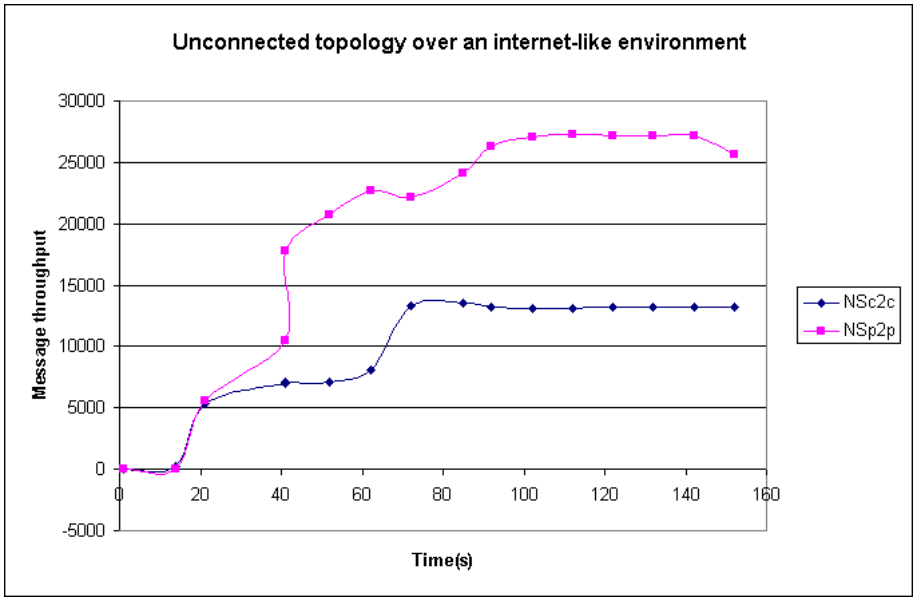

Figure 6.15: The unconnected application topology on Internet-like environments.

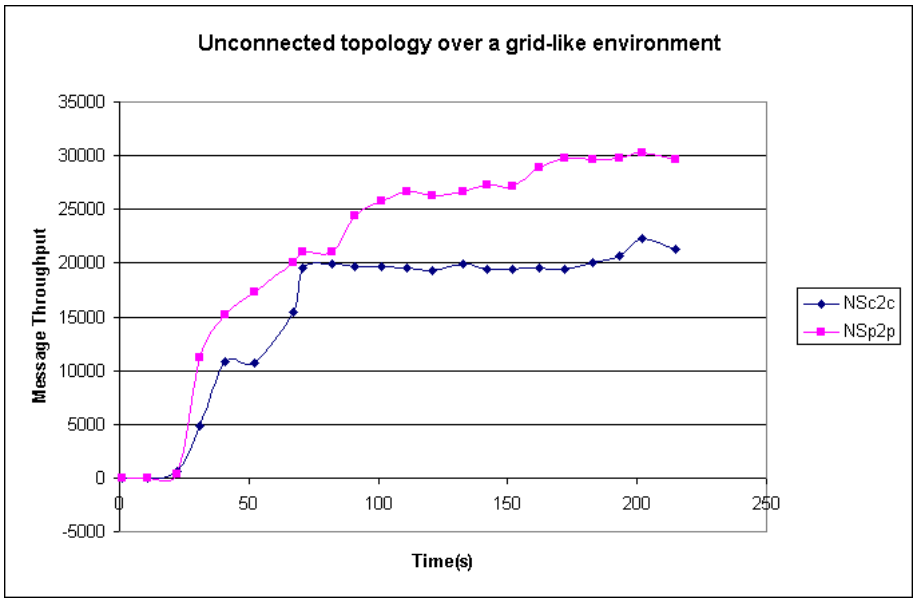

Figure 6.16: The unconnected application topology on Grid-like environments.

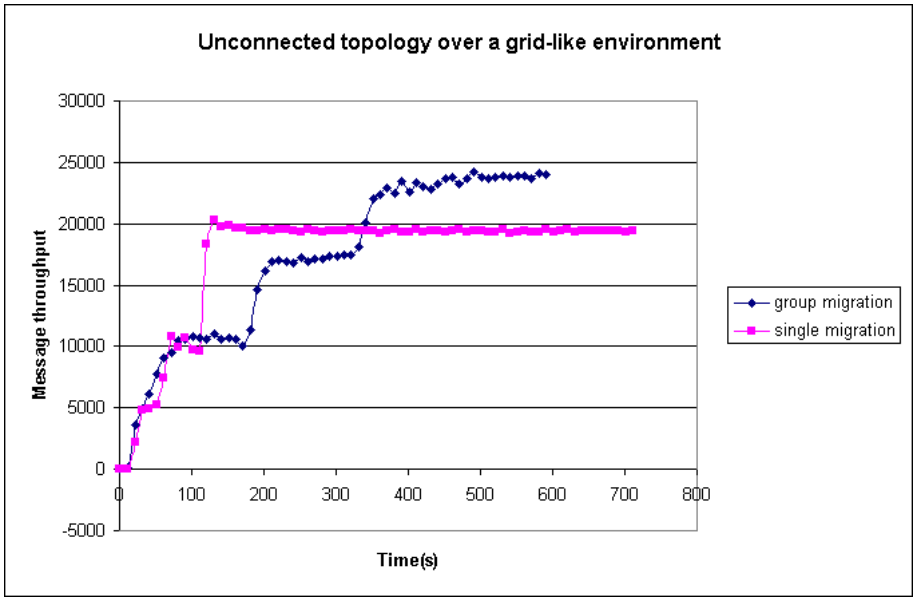

Figure 6.17: Single vs. group migration for the unconnected application topology.

better in Internet-like environments that lack structure for highly synchronized parallel and distributed applications, while the c2c topology is more suitable for grid-like environments that have a rather hierarchical structure.

6.3.4 Single vs. Group Migration

In the NSc2c strategy, intra-cluster load balancing can possibly migrate multiple actors at the same time given the centralized knowledge at the cluster manager. We have evaluated these two strategies over a testbed with two clusters. Each cluster consists of 4 machines. Results show that for the 4 application topologies, group migration performs better than individual migration (see Figures 6.17, 6.18, 6.19 and 6.20). While individual migration is more conservative and results in a more stable behavior of the application throughput as the experiments show, migrating multiple actors simultaneously can balance the load much more quickly.

6.3.5 Overhead Evaluation

The overhead of using IOS with SALSA in a static environment was evaluated using two applications, a massively parallel astronomic application and a tightly

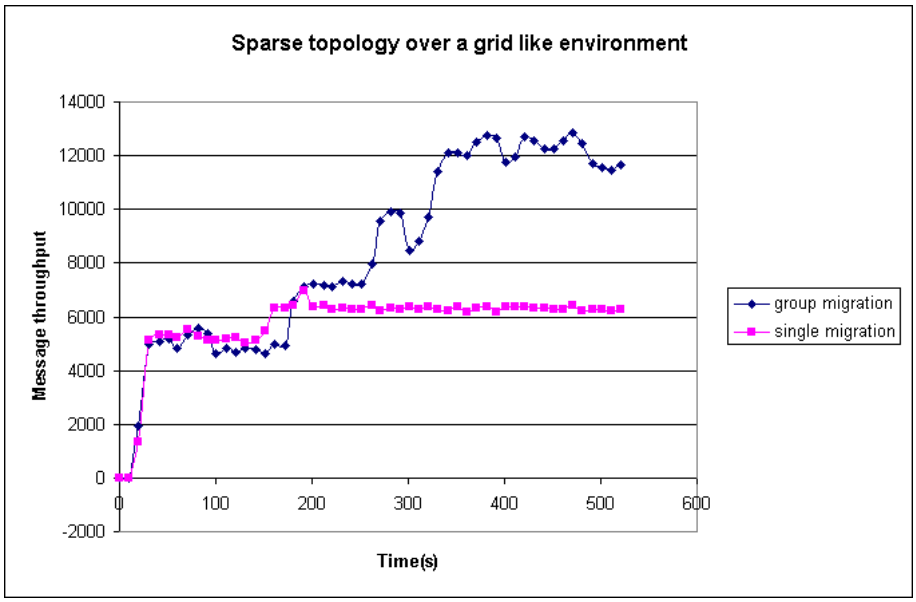

Figure 6.18: Single vs. group migration for the sparse application topology.

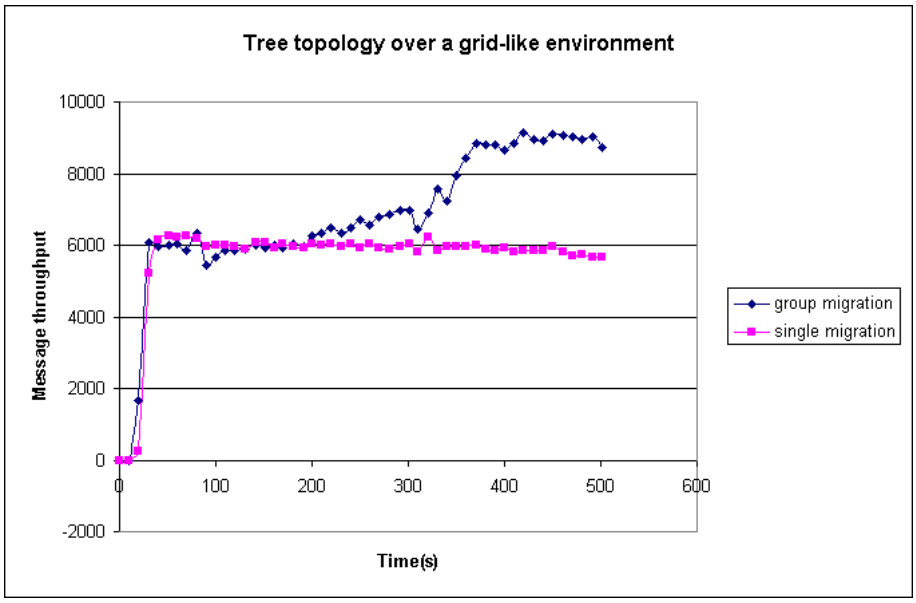

Figure 6.19: Single vs. group migration for the tree application topology.

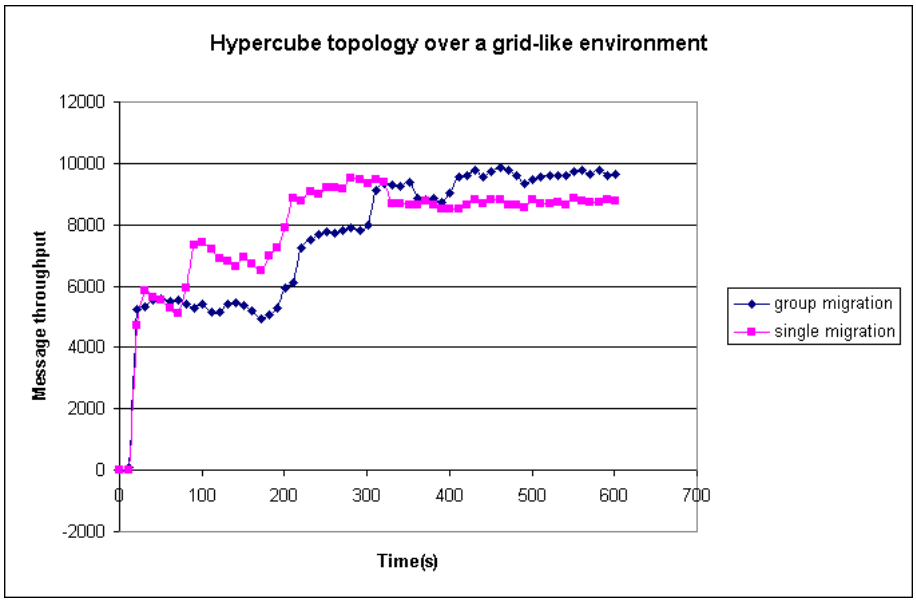

Figure 6.20: Single vs. group migration for the hypercube application topology.

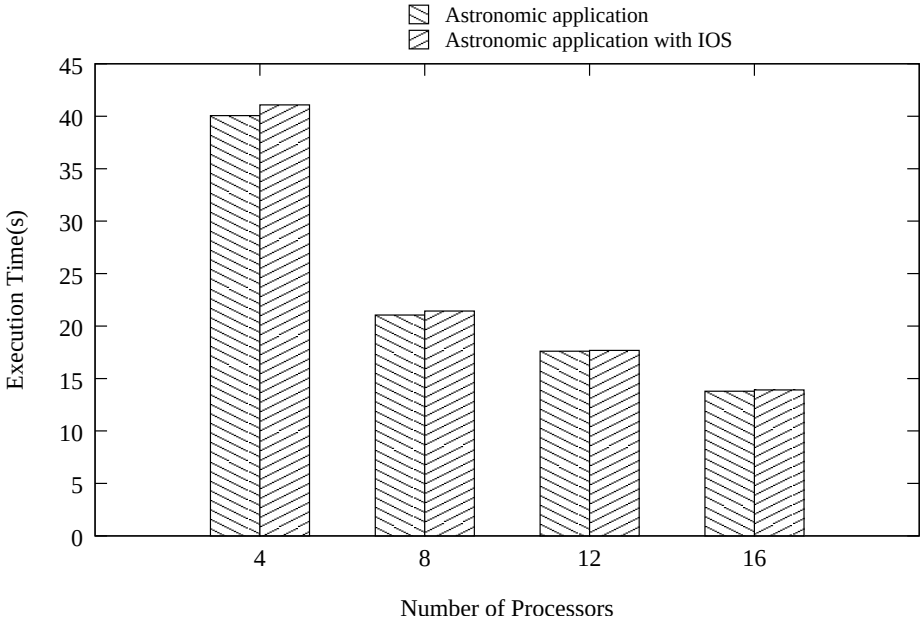

Figure 6.21: Overhead of using SALSA/IOS on a massively parallel astronomic data-modeling application with various degrees of parallelism on a static environment.

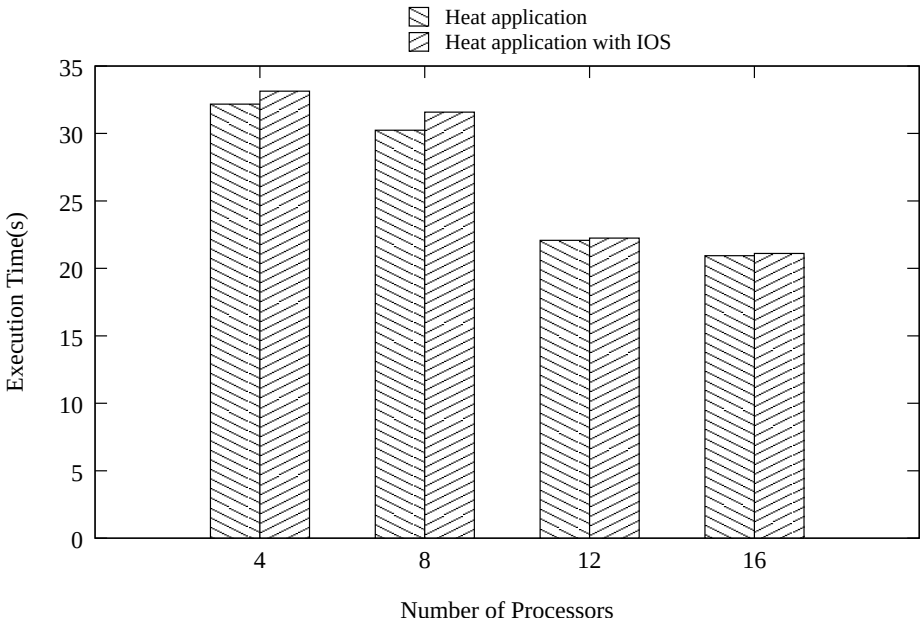

Figure 6.22: Overhead of using SALSA/IOS on a tightly synchronized two-dimensional heat diffusion application with various degrees of parallelism on a static environment.

coupled two-dimensional heat diffusion application. Both applications are iterative. Therefore the average time spent of executing every iteration is a good measure of how well the application is performing. Figures 6.21 and 6.22 show the the time spent by each iteration using SALSA and SALSA/IOS. In both cases the overhead was minimal, around 0.5% for the astronomic application and 2% for the two-dimensional heat diffusion application.

6.4 Adaptation Experiments with Iterative MPI Applications

For the experiments described in this section, we used cluster A and cluster B, which provide a heterogeneous environment of SUN machines with various processing powers and various network capabilities. We also used MPICH2 [1] for comparative purposes, a freely available implementation of the MPI-2 standard.

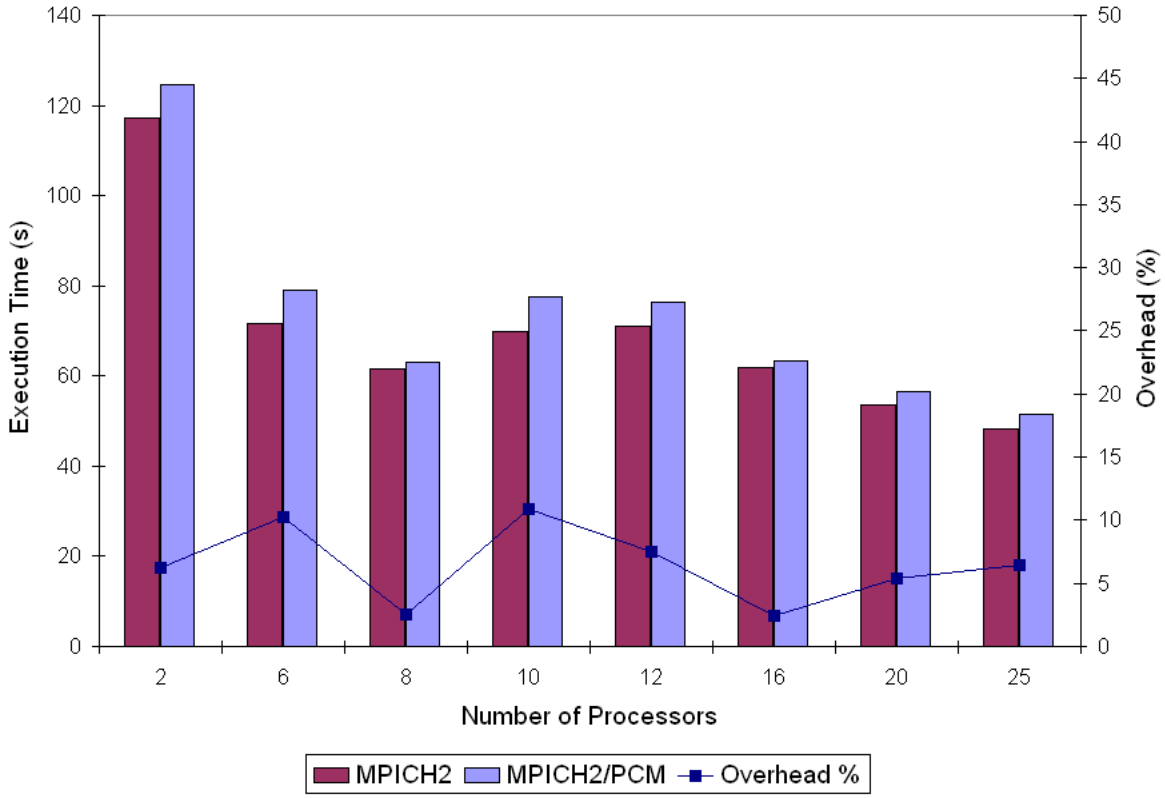

Figure 6.23: Overhead of the PCM library.

6.4.1 Profiling Overhead

To evaluate the overhead of the PCM profiling and status probing, we have run the heat diffusion application with the base MPICH2 implementation and with the PCM instrumentation. We run the simulation with 40 processes on a different numbers of processors. Figure 6.23 shows that the overhead of the PCM library does not exceed 11% of the application's running time. The measured overhead includes profiling, status probing, and synchronization. The library supports tunable profiling, whereby the degree of profiling can be decreased by the user to reduce its intrusiveness.

6.4.2 Reconfiguration Overhead

To evaluate the cost of reconfiguration of the PCM library, we varied the problem data size of the heat diffusion application and measured the overhead of reconfiguration in each case. In the conducted experiments, we started the application on a

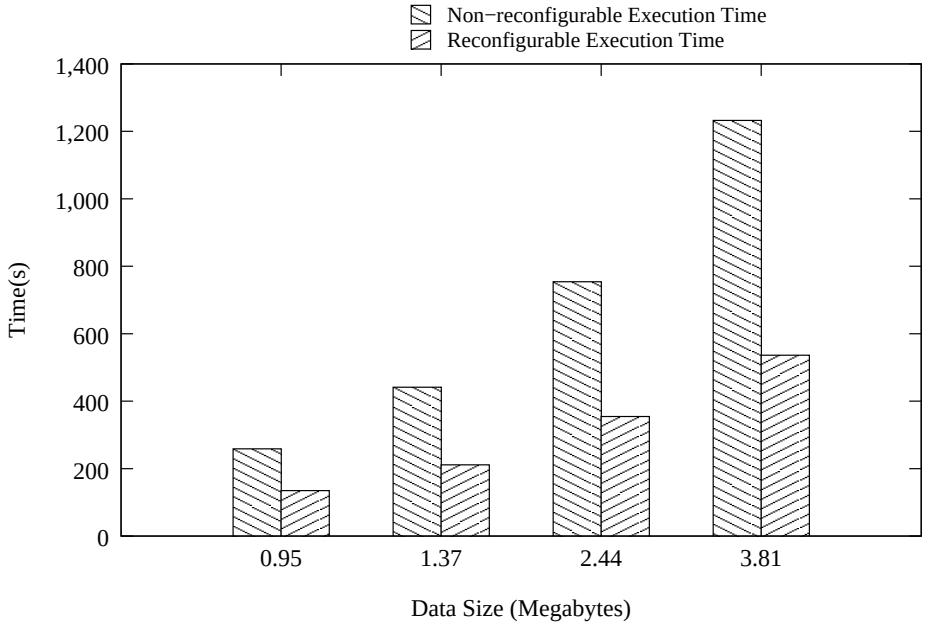

Figure 6.24: Total running time of reconfigurable and non-reconfigurable execution scenarios for different problem data sizes for the heat diffusion application.

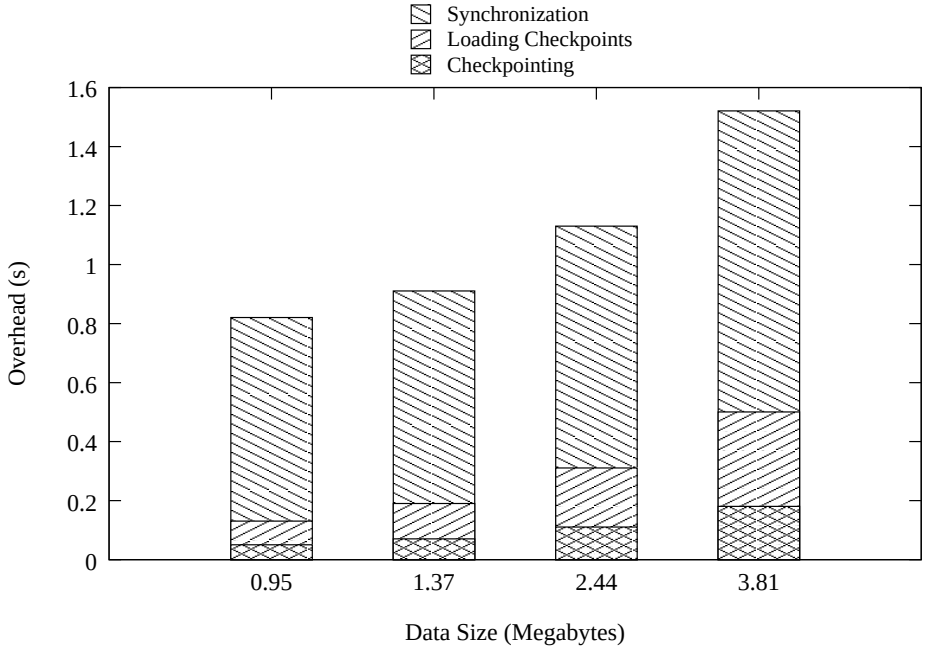

Figure 6.25: Breakdown of the reconfiguration overhead for the experiment of Figure 6.24.

local cluster. We then introduced artificial load in one of the participating machines. One execution was allowed to reconfigure by migrating the suffering process to an available node that belongs to a different cluster, while the second execution was not allowed to reconfigure itself. The experiments in Figures 6.24 and 6.25 show that in the studied cases, reconfiguration overhead was negligible. In all cases, it accounted for less than 1% of the total execution time. The application studied is not data-intensive. We also used an experimental testbed that consisted of 2 clusters that belong to the same institution. So the network latencies were not significant. The reconfiguration overhead is expected to increase with larger latencies and larger data sizes. However, reconfiguration will still be beneficial in the case of large-scale long-running applications. Figure 6.25 shows the breakdown of the reconfiguration cost. The overhead measured consisted mainly of the costs of checkpointing, migration, and the synchronizations involved in re-arranging the MPI communicators. Due to the highly synchronous nature of this application, communication profiling was not used because a simple decision function that takes into account the profiling of the CPU usage was enough to yield good reconfiguration decisions.

6.4.3 Performance Evaluation of MPI/IOS Reconfiguration

6.4.3.1 Migration Experiments

We conducted two experiments using the heat application using the MPI/IOS framework and MPICH2 under similar load conditions. For both experiments, we started running the application, then we emulated a shared and dynamic environment with varying load conditions by introducing artificial load in some of the cluster nodes and varying it periodically.

The first experiment was conducted with MPI/IOS. Figure 6.26 shows the performance of the application. We started running the application with 8 processes on 4 processors. The remaining 4 processors joined the virtual IOS network gradually. We started increasing gradually the load on one of the cluster nodes (two processors) participating in the computation. One node joined the virtual IOS network around iteration 1000 and started sending work-stealing requests. This caused one process to migrate to the new machine and to reduce the load on its original hosting node.

We notice an increase in the application throughput. The load in the slow machine increased even further around iteration 2500. Around iteration 3000, a fourth node joined the virtual network and started sending work-stealing request messages. At this point, two processes migrated to this machine. This caused the slow processors to be eliminated from the computation. The application ended up using a total of the 6 best available processors, which caused a substantial increase in its performance. The total execution time of the application was 645.67s.

Figure 6.26 also shows the performance of the application using MPICH2. We emulated the same load conditions as in the first experiment. With no ability to adapt, the application run over the same hardware configuration and experienced a constant slowdown in its performance. The highly synchronized nature of this application causes it to run as fast as the slowest processor. The application took 1173.79s to finish, about an 81.8% decrease in performance compared to the adaptive execution.

IOS allows the evaluated test application to adapt by trying to migrate all work from slow processors anytime an expected increase in performance is predicted through migration. This evaluation occurs when the work-stealing mechanism gets triggered by a processor becoming lightly-loaded or a new processor becoming available.

6.4.3.2 Split and Merge Experiments

Split/Merge Features. An experiment was setup to evaluate the split and merge capabilities of the PCM malleability library. The heat diffusion application was started initially on 8 processors with a configuration of one process per processor. Then, 8 additional processors at iteration 860 were made available. 8 additional processes were split and migrated to harness the newly available processors. Figure 6.27 shows the immediate performance improvement that the application experienced after this expansion. The sudden drop in the application's throughput at iteration 860 is due to the overhead incurred by the split operation. The collective split operation was used in this experiment because of the large number of resources that have become available. The small fluctuations in the throughput are due to the shared

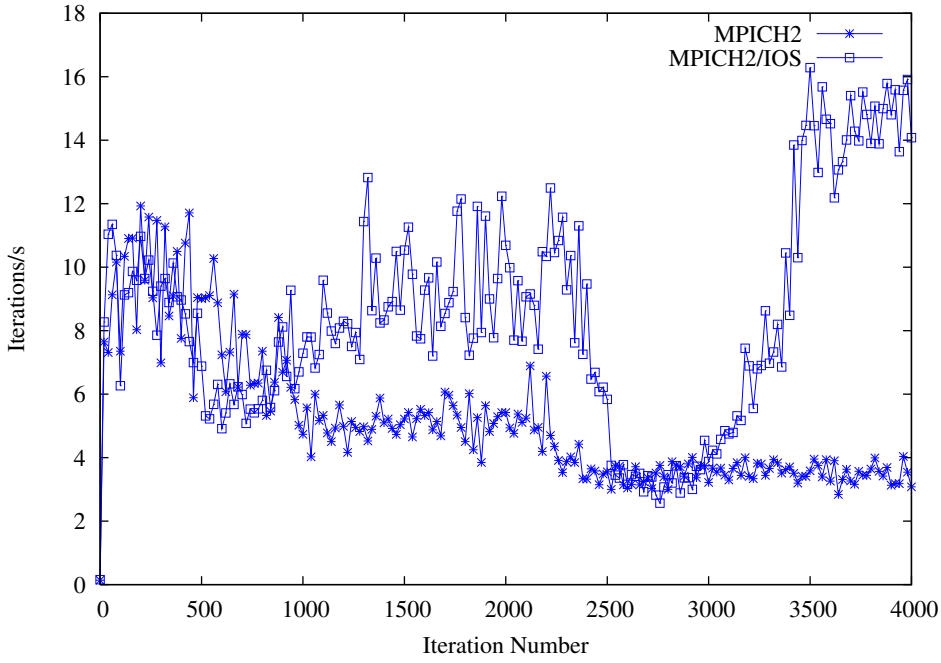

Figure 6.26: Performance of the heat diffusion application using MPICH2 and MPICH2 with IOS.

nature of the cluster used for experiments.

Gradual Adaptation with Malleability and Migration. The following experiment shown in Figure 6.28 illustrates the usefulness of having the 1 to N split and merge operations. When the execution environment experiences small load fluctuations, a gradual adaptation strategy is needed. The heat application was launched on a dual-processor machine with 2 processes. Two binary split operations occurred at events 1 and 2. The throughput of the application decreased a bit because of the decrease of the granularity of the processes on the hosting machine. At event 3, another dual-processor node was made available to the application. Two processes migrated to the new node. The application experienced an increase in throughput as a result of this reconfiguration. A similar situation happened at events 5 and 6, which triggered two split operations, and then two migrations to another dual-processor node at event 7. An increase in throughput was noticed after the migration at event 7 due to a better distribution of work. A node left at event 8 which caused two processes to be migrated to one of the participating machines. A merge operation happened at event

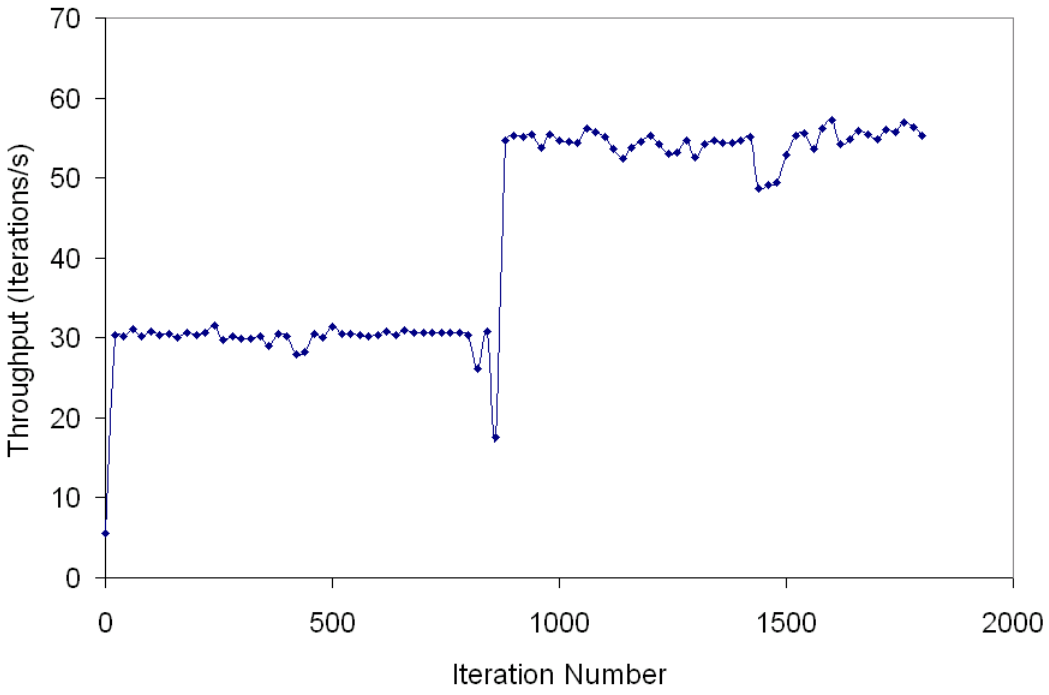

Figure 6.27: The expansion and shrinkage capability of the PCM library.

9 in the node with excess processes, which improved the application's throughput.

Malleability versus Migration The advantage of using both malleability and migration is illustrated through a reconfigurable execution of the astronomy application on a dynamic environment. In one experiment, only migration is used as a reconfiguration strategy, while in another experiment, both malleability and migration are used to adapt the application. The experiments were both run under controlled environments that provide the same runtime behavior for comparative purposes. The time spent per iteration gives a good measure about the relative performance of the application in both scenarios. Figure 6.29 shows the iteration times for the application as the environment changes dynamically. Every 5 iterations, the environment changed by having some processors join or leave the virtual network of IOS agents. Typically, the application is allowed to adapt in one or two iterations, and then the environment remains stable for another 5 iterations. For both tests, the dynamic environment changed from 8 to 12 to 16 to 15 to 10 and then back to 8 processors.

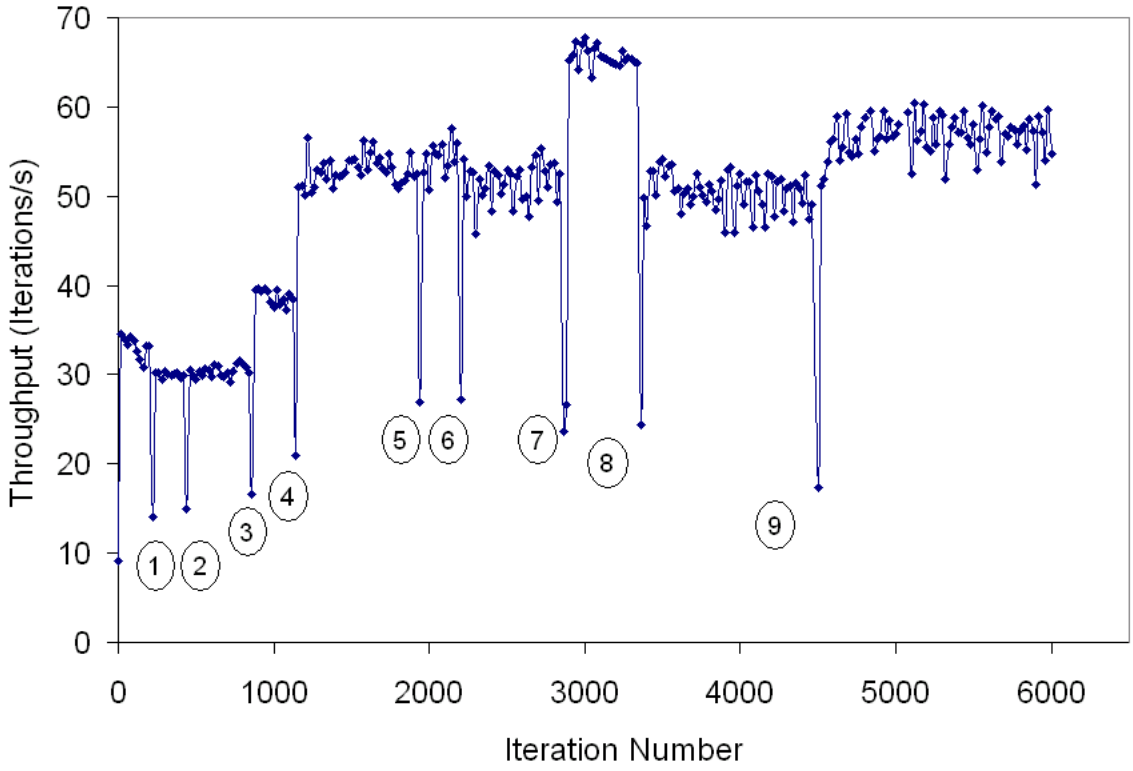

Figure 6.28: Adaptation using malleability and migration as resources leave and join

Malleability allowed a more flexible granularity and hence data distribution, resulting in improved performance when the application was running on 12, 15 and 10 processors. Performance was the same for 8 and 16 processors as migration was able to evenly distribute the entities in both environments. However, for the 12 processor configuration the malleable entities were 6% faster, and for the 15 and 10 processor configurations, malleable entities were 15% and 13% faster respectively. Overall, the astronomy application using both malleability and migration was 5% faster than only using migration.

6.5 Summary and Discussion

The experimental evaluation of IOS has shown that progressively more informed load balancing schemes improve the performance of distributed applications on dy-

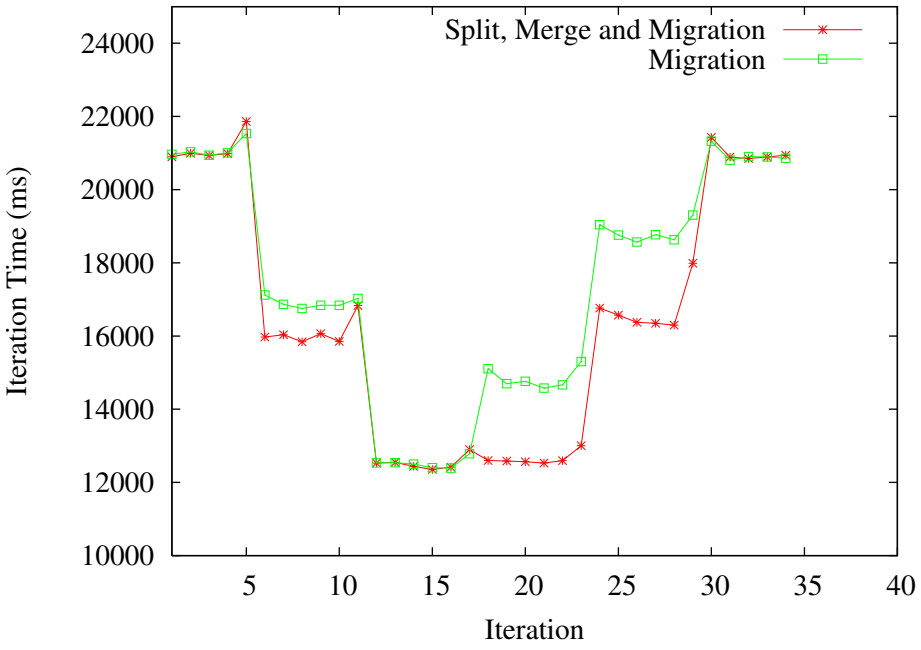

Figure 6.29: Dynamic reconfiguration using malleability and migration compared to dynamic reconfiguration using migration alone in a dynamic virtual network of IOS agents. The virtual network was varied from 8 to 12 to 16 to 15 to 10 to 8 processors. Malleable entities outperformed solely migratable entities on average by 5%.

namic networks. It has also empirically enabled us to evaluate different decentralized coordination strategies on diverse application and physical network topologies. The implemented strategies have focused mainly on (1) profiling CPU processing power in distributed physical machines, (2) being sensitive to application's topologies to attempt co-location of its entities with high frequencies of communication, and (3) being sensitive to the physical network topologies to attempt minimization of communications over links with high latency and low bandwidth. Since applications have different resource requirements, the ideal level of profiling is highly dependent on the nature of computation that is being performed. For this reason, a modular architecture that enables plugging-in components for different decision, profiling, and coordination strategies has been developed.

Applications with diverse communication patterns were evaluated on two rep-

resentative physical networks: first, an *Internet/Web computing* environment with highly diverse computation and communication capabilities across its nodes; and second, a *Grid computing* environment with more structured topology: a set of tightly-coupled and homogeneous clusters. Our results show that with applications exhibiting high communication to computation ratios (e.g., hypercube and tree application topologies), the p2p agent topology performs better on Internet-like environments while the c2c agent topology performs better on Grid-like environments. On sparse application communication topologies, c2c outperforms p2p on both Internet- and Grid-like environments, while on unconnected application topologies (representing massively parallel applications), p2p outperforms c2c in both Internet- and Grid-like environments. Using group migration to balance the distribution of load across several nodes performed empirically better than using single migration of actors.

The experimental evaluation of the MPI implementation of the iterative heat diffusion problem has demonstrated the importance of augmenting long-running MPI application with reconfiguration capabilities to achieve high performance. We have measured the middleware overhead in static environments demonstrating that it is less that 7% on average, yet reconfiguration on dynamic environments can lead to significant improvement in application execution time. Our performance evaluation has also demonstrated the usefulness of malleable operations in improving the performance of iterative applications in dynamic environments where resources join and leave. Collective malleable operations are more appropriate in dynamic environments with large load fluctuations, while individual split and merge operations are more appropriate in environments with small load fluctuations.

CHAPTER 7

Conclusions and Future Work

We have proposed, designed, and implemented a framework for dynamic middleware-triggered application reconfiguration. The Internet Operating System middleware provides a virtual execution environment that hides complex resource management and reconfiguration issues from high-level applications. IOS has been designed with the following key characteristics: 1) **Architecture Modularity** to allow for extensible and pluggable reconfiguration mechanisms to accommodate various execution environments and different applications, 2) **Generic Interfaces** to allow various programming models and technologies to leverage IOS dynamic reconfiguration mechanisms, 3) **Decentralized Strategies** to achieve scalable decisions, and 4) **Adaptability to Dynamic Environments** to allow applications to adjust their resource allocations following the availability of resources. IOS allows fine-grained reconfiguration, whereby processes migrate to harness better resources or adjust their granularity to scale up or shrink down the applications as necessary.

This dissertation also explored several mechanisms for addressing some of the functional issues at the application-level to transform applications into reconfigurable ones. Such issues include migration and malleability. We have focused mainly on message passing applications, following the Message Passing Interface (MPI) standard, due to its importance and wide adoption by the scientific and engineering communities. We presented several extensions to MPI to allow applications to become reconfigurable. The Process Checkpointing, Migration, and Malleability (PCM) library is a user-level library that provides checkpointing, profiling, migration, and

malleability capabilities for a large class of iterative applications. PCM implements IOS generic profiling and reconfiguration interfaces, and therefore enables MPI applications to benefit from IOS reconfiguration policies. With minimal code modification, a PCM-instrumented MPI application becomes malleable and ready to be reconfigured transparently by the IOS middleware. In addition, legacy MPI applications can benefit tremendously from the reconfiguration features of IOS by simply inserting a few calls to the PCM API. The practical implementation of the PCM library has been validated using a realistic fluid dynamics application that models heat diffusion over a solid and other representative iterative MPI applications.

IOS is currently being used as an experimental testbed for two multi-disciplinary scientific applications. The first application is in the area of astrophysics with the goal of analyzing data from the Sloan Digital Sky Survey to find structure in the outer layers of the Milky Way. The second one is an application in nuclear physics that performs partial wave analysis to discover missing baryons, sub-atomic particles predicted by theory that have not been experimentally observed.

This dissertation represents a step in ongoing research efforts to realize the vision of efficient, seamless, and easy deployment of applications in dynamic grid environments. There are still several issues worthy of future investigations. The remaining sections discuss possible future directions.

7.1 Other Application Models

The PCM library has been designed to work with iterative applications. In addition, the split and merge library components have been designed to work with applications that have regular distribution of data. This work can be extended to handle other application and data distribution models. For the case of non-iterative applications, notifying applications about a specific reconfiguration request and getting all the processes to synchronize is a challenging task. Another challenge is being able to notify MPI processes asynchronously. The current architecture relies on polling the underlying middleware. Callback mechanisms provide an alternative to allow for asynchronous notifications.

The current PCM extensions do not consider multi-threaded MPI environments.

Multi-threading has several advantages even on a single machine, namely greater concurrency and the ability to mask memory latencies and achieve greater CPU utilization. Multi-threading has become even more important with the emergence of multi-core architecture and the need for fine-grained concurrency control. Future work includes exploring the challenges involved in reconfiguring multi-threaded and non-regular applications and integrating them with our reconfiguration framework.

There is currently a great interest within the grid community toward service-oriented computing (SOC). A service defines a behavior that is provided by a particular software component and that adheres to a set of standards to allow for its discovery, query and use by other components. A service-oriented architecture emphasizes interoperability and dynamic discovery. The SOC model is attractive to grid users because it provides high-level abstractions that facilitate access to computational resources, storage resources, and network resources in a unified way. Applications can be developed by selecting and composing different services. Adopting the SOC model in grid environments requires adaptation because of the dynamic behavior of grids and the rapidly evolving nature of services. In addition to monitoring resource usage and service performance, metadata of services should be monitored to be able to reason about when it is appropriate to collocate certain services, compose them, or decompose them. Business applications have also different characteristics from scientific and engineering applications. They have a highly transactional nature and are not usually computationally intensive. The key optimization goal of such applications is response time. IOS policies do not address the features of business applications. Future work includes investigating new reconfiguration policies that target specifically the needs and nature of SOC.

7.2 Large-scale Deployment and Security

The performance experiments that were conducted to evaluate the reconfiguration protocols of IOS have been done using clusters at RPI within the same administrative domain. The scale of these clusters ranges from small to medium. Future work should focus on stress-testing IOS capabilities on much larger and more dynamic environments such as PlanetLab [31] and TeraGrid [86]. PlanetLab provides a testbed

for Internet-scale applications. It consists of inter-networked machines located at sites around the world.

Security becomes a major challenge in open distributed systems. Security issues have not been addressed in the current implementation of IOS. One possible solution is to leverage existing research in grid security, such as Globus security infrastructure [47]. Future work may extend IOS to IOS-G which combines IOS dynamic reconfiguration mechanisms with Globus protocols for security, and inter-domain resource discovery, access, and management.

7.3 Replication as Another Reconfiguration Strategy

Computational grids are mainly characterized by having computationally intensive applications that usually operate on small data sets. However, in data grids [84], data is considered a first class citizen and applications are both highly computational and data intensive. In such cases, optimizing data access is key to ensure efficient deployment in large grid environments. Replication is considered one of the major optimization techniques for providing fast data access [15, 68].

We investigated process migration, split and merge as possible ways of reconfiguring applications in dynamic grid environments. Another natural direction is to use process or data replication as another reconfiguration strategy. Replication strategies can improve the performance, availability, and fault-tolerance of applications. New replication reconfiguration strategies need to be designed, developed and integrated with the IOS middleware. Middleware can dynamically profile data access patterns and decide when and where to replicate data.

7.4 Scalable Profiling and Measurements

Application-level and resource-level profiling are key to determining how to adapt applications dynamically to the constantly changing resources of dynamic grids. However, profiling imposes a performance penalty on applications. Also communicating large amounts of profiled data to the middleware can consume a large amount of computational and communication resources. Statistical and data mining methods

can be used to extract useful patterns from the gathered profiled information and approximate global knowledge for less intrusive and more scalable profiling techniques.

7.5 Clustering Techniques for Resource Optimization

When new resources join or existing resources leave, middleware services need to collaborate with the application to reconfigure it appropriately to utilize the new available resources or migrate away from the slow or leaving resources. In many instances, it makes more sense to perform collective reconfiguration of a group of application's entities. Clustering the application communication graph can help determine potential groups for reconfiguration. One direction is to explore applying graph clustering techniques to the application graph. In a largely distributed environment that relies on decentralized coordination techniques, having access to the full graph of the application is not feasible. Clustering techniques have mainly been studied on full graphs. Future work includes developing new algorithms for partial graph clustering and studying their composability.

7.6 Automatic Programming

Automatic programming is a future direction for developing grid applications. Existing or new code analysis techniques and automatic code generation techniques can be used to transparently modify existing applications into reconfigurable ones. Another interesting research direction is to come up with high-level specifications or scripting languages that allow developers to outline the functionality needed in high-level terms. Then, automatic programming tools can translate such descriptions into specialized grid-aware and reconfigurable code. This will ease the development and deployment of applications in grid environments.

7.7 Self-reconfigurable Middleware

Empirical results have shown that different middleware profiling, decision and communication strategies perform better for different applications and physical network types. In addition to reconfiguring applications due to changes in application resource consumption and communication patterns and the dynamic physical network,

the IOS middleware should reconfigure itself—autonomously selecting different profiling and decision strategies depending on its environment. Likewise, the middleware agent topology should adapt to minimize the overhead of inter agent communication and improve the application reconfiguration decision process.

References

[1] Argone National Laboratory, MPICH2, `http://www-unix.mcs.anl.gov/mpi/mpich2`.

[2] The KaZaA home page, `http://www.kazaa.com/us/index.htm`.

[3] The Napster home page, `http://www.napster.com/`.

[4] Sloan Digital Sky Survey, `http://www.sdss.org/`.

[5] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: a tool for performing parametrised simulations using distributed workstations. In *HPDC '95: Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing - HPDC '95*, pages 520–528, Washington, DC, USA, 1995. IEEE Computer Society.

[6] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, 2003.

[7] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[8] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III*, pages 345–363. International Federation of Information Processing Societies (IFIP), Elsevier Science Publisher, 1993.

[9] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[10] G. Agha and C. Varela. Worldwide computing middleware. In M. Singh, editor, *Practical Handbook on Internet Computing*, pages 38.1–21. CRC Press, 2004.

[11] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with dynamic resource selection and allocation in a grid environment. *International Journal of High-Performance Computing Applications*, 15(4):345–358, 2001.

[12] G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, page 25pp., New York, NY, USA, 2001. ACM Press.

[13] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[14] P. Asadzadeh, R. Buyya, C. L. Kei, D. Nayar, and S. Venugopal. *High Performance Computing: Paradigm and Infrastructure*, chapter Global Grids and Software Toolkits: A Study of Four Grid Middleware Technologies, pages 431–455. Wiley Press, New Jersey, USA, June 2005.

[15] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. Simulation of dynamic grid replication strategies in OptorSim. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 46–57, London, UK, 2002. Springer-Verlag.

[16] F. Berman and R. Wolski. The AppLeS Project: A Status Report. In *Proceedings of the 8th NEC Research Symposium*, page 19pp., Berlin, Germany, 1997.

[17] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003.

[18] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, page 28pp., 1996.

[19] M. A. Bhandarkar, L. V. Kalé, E. de Sturler, and J. Hoeflinger. Adaptive load balancing for MPI programs. In *Proceedings of the International Conference on Computational Science-Part II*, pages 108–117. Springer-Verlag, 2001.

[20] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. volume 2, pages 39–59, New York, NY, USA, 1984. ACM Press.

[21] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[22] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

[23] J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'89)*, pages 109–129. Cambridge University Press, 1989.

[24] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, New York, NY, USA, 1981. ACM Press.

[25] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture of a resource management and scheduling system in a global computational grid. *High Performance Computing in the Asia-Pacific Region*, 1:283–289, 2000.

[26] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. Maan: A multi-attribute addressable network for grid information services. In H. Stockinger, editor, *GRID*, pages 184–191. IEEE Computer Society, 2003.

[27] H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[28] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 75–76, Washington, DC, USA, 2000. IEEE Computer Society.

[29] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw. The Legion resource management system. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.

[30] A. A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *J. Parallel Distrib. Comput.*, 63(5):597–610, 2003.

[31] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[32] K. Clark, J. E. Flaherty, and M. S. Shephard. *Applied Numerical Mathematics, special ed. on Adaptive Methods for Partial Differential Equations*, 14, April 1994.

[33] Clip2.com. The gnutella protocol specification v0.4, `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`, 2000.

[34] S. Cluet, O. Kapitskaia, and D. Srivastava. Using LDAP directory caches. In *PODS '99: Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 273–284, New York, NY, USA, 1999. ACM Press.

[35] N. Coleman, R. Raman, M. Livny, and M. Solomon. Distributed policy management and comprehension with classified advertisements. Technical Report UW-CS-TR-1481, University of Wisconsin - Madison Computer Sciences Department, April 2003.

[36] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE Symposium On High Performance Distributed Computing*, pages 181–194, 2001.

[37] T. Desell, K. E. Maghraoui, and C. Varela. Malleable components for scalable high performance computing. In *Proceedings of HPDC'15 Workshop on HPC Grid programming Environments and Components (HPC-GECO/CompFrame)*, pages 37–44, Paris, France, June 2006. IEEE Computer Society.

[38] R. Elsasser, B. Monien, and R. Preis. Diffusive load balancing schemes on heterogeneous networks. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 30–38. ACM Press, 2000.

[39] D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[40] A. C. F. Berman, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS project: Software support for high-level grid application development. *International Journal of High-Performance Computing Applications*, 15(4):327–344, 2001.

[41] M. Feeley and J. S. Miller. A parallel virtual machine for efficient scheme compilation. In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 119–130, New York, NY, USA, 1990. ACM Press.

[42] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In D. G. Feitelson and L. Rudolph, editors, *JSSPP*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 1996.

[43] J. Ferber and J. Briot. Design of a concurrent language for distributed artificial intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

[44] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Applied Numerical Mathematics*, 26:241–263, 1998.

[45] I. Foster and C. Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[46] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[47] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[48] I. T. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *IPTPS*, pages 118–128, 2003.

[49] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[50] W. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, pages 530–533. IEEE Computer Society, 1995.

[51] A. Gupta, A. Tucker, and L. Stevens. Making effective use of shared-memory multiprocessors: the process control approach. Technical Report CSL-TR-91-475, Stanford, CA, USA, 1991.

[52] R. H. Halstead-Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 9–17, New York, NY, USA, 1984. ACM Press.

[53] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[54] C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.

[55] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, Aug. 2002.

[56] W. Hoschek, F. J. Jaén-Martínez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In R. Buyya and M. Baker, editors, *GRID*, volume 1971 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2000.

[57] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, pages 306–322, College Station, Texas, October 2003.

[58] C. Huang, G. Zheng, L. Kalé, and S. Kumar. Performance evaluation of adaptive MPI. In *PPoPP '06: Proceedings of Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 12–21, New York, NY, USA, 2006. ACM Press.

[59] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.

[60] K. Jun, L. Bni, K. Palacz, and D. C. Marinescu. Agent-based resource discovery. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, pages 43–52, Washington, DC, USA, 2000. IEEE Computer Society.

[61] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. *J. Parallel Distrib. Comput.*, 63(5):551 – 563, 2003.

[62] A. R. Karthik. Load balancing in structured p2p systems. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, page 6pp., Berkeley, CA, 2003.

[63] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, page 39pp., 1995.

[64] F. Kon, R. H. Campbell, M. D. Mickunas, K. Nahrstedt, and F. J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, August 2000.

[65] F. Kon, J. R. Marques, T. Yamane, R. H. Campbell, and M. D. Mickunas. Design, implementation, and performance of an automatic configuration service for distributed component systems. *Softw., Pract. Exper.*, 35(7):667–703, 2005.

[66] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 32(2):135–164, 2002.

[67] T. T. Kwan and D. A. Reed. Performance evaluation of an infrastructure for worldwide parallel computing. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 379–386, Washington, DC, USA, 1999. IEEE Computer Society.

[68] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman. Simulation of dynamic data replication strategies in data grids. In *Proceedings of the 12th Heterogeneous Computing Workshop*, page 8pp., 2003.

[69] G. Lanfermann, G. Allen, T. Radke, and E. Seidel. Nomadic migration: A new tool for dynamic grid computing. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, pages 429–430. IEEE Computer Society, 2001.

[70] G. Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13:645–662, 2001.

[71] W. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *SIGMETRICS '86/PERFORMANCE '86: Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modeling, Measurement and Evaluation*, pages 54–69, New York, NY, USA, 1986. ACM Press.

[72] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[73] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 20(4):467–480, 2006.

[74] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, pages 21–32, Feb. 2003.

[75] C. Mastroianni, D. Talia, and O. Verta. A super-peer model for resource discovery services in large-scale grids. *Future Gener. Comput. Syst.*, 21(8):1235–1248, 2005.

[76] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11(2):146–178, 1993.

[77] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall/Winter 1994.

[78] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface `http://nf.apac.edu.au/training/MPI/MPI_2.0/mpi2-report.html`. Technical report, 1996.

[79] R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[80] A. Natrajan, M. Humphrey, and A. Grimshaw. Grids: Harnessing geographically-separated resources in a multi-organisational context. In *Proceedings of High Performance Computing Systems*, pages 319 – 339, 2001.

[81] Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment, 1998. Work in Progress. http://osl.cs.uiuc.edu/foundry/.

[82] O. Othman and D. C. Schmidt. Issues in the Design of Adaptive Middleware Load Balancing. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, pages 205–213, Snow Bird, Utah, USA, 2001.

[83] D. Puppin, S. Moncelli, R. Baraglia, N. Tonellotto, and F. Silvestri. A grid information service based on peer-to-peer. In *11th Euro-Par conference*, volume 3648 of *LNCS*, pages 454–464. Springer, 2005.

[84] A. Rajasekar, M. Wan, R. Moore, G. Kremenek, and T. Guptil. Data grids, collections, and grid bricks. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, pages 2–9, Washington, DC, USA, 2003. IEEE Computer Society.

[85] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, pages 161–172, 2001.

[86] D. A. Reed. Grids, the teragrid, and beyond. *Computer*, 36(1):62–68, 2003.

[87] S. Ren, G. A. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36:4–12, 1996.

[88] K. A. Robbins and S. Robbins. *The Cray X-MP/Model 24, A Case Study in Pipelined Architecture and Vector Processing*, volume 374 of *Lecture Notes in Computer Science*. Springer, 1989.

[89] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *SPAA '91: Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, New York, NY, USA, 1991. ACM Press.

[90] H. Saito, Y. Miyao, and M. Yoshida. Traffic engineering using multiple multipoint-to-point LSPs. In *INFOCOM (2)*, pages 894–901, 2000.

[91] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In *HPCN Europe*, pages 491–502, 1997.

[92] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A remote procedure call API for grid computing. In *Grid Computing Workshop*, pages 274–278, Baltimore, MD, 2000.

[93] O. Sievert and H. Casanova. A simple MPI process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 18(3):341–352, 2004.

[94] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 141–148, New York, NY, USA, 1998. ACM Press.

[95] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531. IEEE Computer Society, 1996.

[96] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[97] D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996. TR UIUCDCS-R-96-1950.

[98] R. Sylvia, F. Paul, H. Mark, K. Richard, and S. Scott. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 31, pages 161–172. ACM Press, October 2001.

[99] D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 07(4):96, 94–95, 2003.

[100] D. Talia and P. Trunfio. Adapting a pure decentralized peer-to-peer protocol for grid services invocation. *Parallel Processing Letters (PPL)*, 15(1-2):67–84, 2005.

[101] K. Taura, K. Kaneda, and T. Endo. Phoenix: a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. In *Proceedings of PPoPP*, pages 216–229. ACM, 2003.

[102] J. D. Teresco, K. D. Devine, and J. E. Flaherty. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chapter Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations, pages 55–81. Springer-Verlag, 2005.

[103] J. D. Teresco, J. Faik, and J. E. Flaherty. Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science and Engineering*, 07(2):40–50, 2005.

[104] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[105] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, 2005.

[106] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in Carnot. *IEEE Parallel and Distributed Technology*, 1(2):16–20, May 1993.

[107] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards high performance peer-to-peer content and resource sharing systems. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, pages 341–355, Pacific Grove, California, USA, 2003.

[108] G. Utrera, J. Corbalán, and J. Labarta. Implementing malleability on mpi jobs. In *IEEE PACT*, pages 215–224. IEEE Computer Society, 2004.

[109] S. Vadhiyar and Dongarra. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[110] S. S. Vadhiyar and J. J. Dongarra. SRS - a framework for developing malleable and migratable parallel applications for distributed systems. In *Parallel Processing Letters*, volume 13, pages 291–312, June 2003.

[111] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Computing*, pages 52–63, July 1998.

[112] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP '01: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 34–43, New York, NY, USA, 2001. ACM Press.

[113] C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, 2001.

[114] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, Dec. 2001. http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf.

[115] C. A. Varela, P. Ciancarini, and K. Taura. Worldwide computing: Adaptive middleware and programming technology for dynamic Grid environments. *Scientific Programming Journal*, 13(4):255–263, December 2005. Guest Editorial.

[116] D. B. Weatherly, D. K. Lowenthal, M. Nakazawa, and F. Lowenthal. Dyn-mpi: Supporting mpi on medium-scale, non-dedicated clusters. *J. Parallel Distrib. Comput.*, 66(6):822–838, 2006.

[117] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998.

[118] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Comput. Syst.*, 15(5-6):757–768, October 1999.

[119] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, Jan. 2004.