A Framework for the Development of Wide Area Distributed Applications

A Thesis
Presented to
The Academic Faculty

by

Brad Byer Topol

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
June, 1998

A Framework for the Development of Wide Area Distributed Applications

Approved:

Mustaque Ahamad, Co-Chairman

John T. Stasko, Co-Chairman

Richard M. Fujimoto

Kishore Ramachandran

Vaidy Sunderam

Date Approved

Dedication

This thesis is dedicated in loving memory to Jules Freedman, Morris Topol, and Selma Topol.

Acknowledgments

I am forever grateful to Professor Mustaque Ahamad for providing me with excellent guidance and innumerable helpful discussions regarding this work, and for showing me that research can be a fun and rewarding experience. I also thank Professor John Stasko for his guidance and many helpful discussions regarding this thesis.

I would also like to thank Professors Vaidy Sunderam and Bevan K Youse. The mentoring and instruction I received from them while an undergraduate at Emory University were excellent preparation for the rigors of a Ph.D. program.

I owe a great deal of thanks to my friends Chris Carothers, Phyllis Schneck, Colleen Kehoe, Drew Kessler, Don Allison, and Peter Lindstrom who were always willing to take time out to talk with me whenever I was bored and for making the long hours spent at Georgia Tech a more pleasant experience.

I would like to thank my grandmother, Sylvia Freedman, for her support and encouragement and for lifting my spirits by referring to me as "Doctor Topol". I also would like to thank my sister Marci and her husband Rick for their support and encouragement and for the wonderful Thanksgiving holidays we shared together.

I thank my fiance, Janet, for her love, support, and patience during this process and for always being there for me.

I thank my parents, Harold and Mady, for their love and support and also for providing me with the opportunities in life that enabled me to pursue my Ph.D.

Contents

List of Tables

List of Figures

Summary

The growth in popularity of the World Wide Web has resulted in the development of a new generation of tools tailored to Internet computing activities. Prominent examples include the Java programming language and Java capable Web browsers. These Web spinoffs are having a profound impact on the field of distributed computing. Whereas distributed computing has traditionally focused on improving the functionality of local clusters of computers, technology is progressing such that wide area computing networks are now becoming a popular target environment for research in distributed computing.

With wide area distributed computing environments, geographically distributed resources such as workstations, personal computers, supercomputers, graphic rendering engines, and scientific instruments will be available for use in a seamless fashion by parallel applications. Many envision that it will be possible to transport application code to remote sites in the wide area virtual computer where it may be executed in the presence of needed resources. The area of research devoted to bringing this vision to reality in the context of scientific applications is referred to as metacomputing.

Metacomputing environments are useful for a variety of distributed and parallel applications, particularly those which need access to remote resources or applications that are able to effectively utilize a substantial number of computing resources that the Internet may easily provide. Moreover, other wide area distributed computing domains such as electronic commerce home service applications require more advanced capabilities than those provided by a standard web browser. Such capabilities must address issues such as heterogeneity, object sharing, and failure resilience in wide area environments.

In this thesis, we investigate the design and implementation of a Java based middleware infrastructure system to enable wide area applications. We then provide an empirical evaluation of our prototype system for local area, wide area, and home service network environments. We also illustrate the system's ability to support the development of several classes of applications which include electronic commerce home service applications, failure resilient scientific applications, and traditional computer supported cooperative work applications. Finally, we present a design for the monitoring and visual presentation of activities associated with wide area distributed computing.

Chapter 1

Introduction

The growth in popularity of the World Wide Web (WWW) has resulted in the development of a new generation of tools tailored to Internet computing activities. Prominent examples include the Java programming language and Java capable Web browsers. These Web spinoffs are having a profound impact on the field of distributed computing. Whereas distributed computing has traditionally focused on improving the functionality of local clusters of computers, technology is progressing such that wide area computing networks are now becoming a popular target environment for research in distributed computing.

With wide area distributed computing environments, geographically distributed resources such as workstations, personal computers, supercomputers, graphic rendering engines, and scientific instruments will be available for use in a seamless fashion by parallel applications[27, 20]. Many envision that it will be possible to transport application code to remote sites in the wide area virtual computer where it may be executed in the presence of needed resources. The area of research devoted to bringing this vision to reality in the context of scientific applications is referred to as *metacomputing*. A metacomputing system is the software infrastructure that orchestrates collections of hosts and networks into functioning as a large virtual computer for use by parallel and distributed applications. These systems are particularly useful when local workstation clusters are not appropriate for the execution of the distributed or parallel application. This can occur if the application requires access to a remote resource such as data from a remote microscope that is not available in the local workstation cluster. Moreover, the application may require more computing resources than are available locally and thus would benefit from the vast amount of remote computing resources available via the Internet.

Another application domain that benefits from advances made in wide area computing is collaborative interactive distributed applications that provide services to the home. Homes are becoming better connected to each other as telephone companies, Internet service providers, and cable companies compete to bring increased bidirectional bandwidth to the home that is capable of supporting interactive applications. Similarly, several home platform options are also becoming available and choices include the PC, network computer, and the settop box. In particular, electronic commerce applications in the home require functionality similar to that required by distributed and parallel applications in a wide area computing environment.

Electronic commerce applications, such as those that facilitate collaborative home shopping, are a prime example of interactive applications that would benefit from the functionality provided by a wide area computing environment. In these types of applications, consumers in different geographical locations can shop together in "virtual stores". These virtual stores can be 2D or 3D. In the virtual stores, each consumer can quickly acquire his or her peers' opinions of the items the consumer is considering purchasing. Group members may locate and suggest items that others in the group have overlooked. Essentially, the consumers are able to maintain in the virtual environment shared access to information as well as the high level of interaction that exists naturally when shopping together in a real store. Shared information access and interactivity are facilities that are not well supported by the mechanisms available in wide area computing environments. Thus, collaborative applications such as these are examples of wide area applications that are poised to become reality once wide area computing environments supporting these mechanisms become available.

The power and benefits of wide area computing do not come without a price, however. Wide area applications by their very nature are distributed applications. As such, they are dependent on some mechanism for state sharing across workstations to allow processors to cooperatively work together. Furthermore, writing and debugging a distributed application is a complex task; much more difficult than writing a sequential program. In many cases this added complexity results from distributed tasks working in a cooperative fashion, a model that is not familiar to traditional sequential programmers. Finally, failures in a wide area computing environment are a relatively common occurrence. Failures not only result from network and workstation problems but also take place when a user at a remote site reboots a machine or kills a remotely executing task; or when a home consumer becomes disinterested in a electronic commerce activity and removes himself or herself from the activity.

In this thesis, we investigate the design and implementation of a Java based middleware infrastructure system to enable wide area applications. We then provide an empirical evaluation of our prototype system for local area, wide area, and home service network environments. We also illustrate the system's ability to support the development of several classes of applications from domains that include electronic commerce home service applications, failure resilient scientific applications, and traditional computer supported cooperative work applications. Finally, we present a design for the monitoring and visual presentation of activities associated with wide area distributed computing.

Contributions

The following are the contributions of our work in the development of system support to enable applications in the wide area environment:

- The Mocha system, which is a wide area computing infrastructure that provides basic facilities such as remote evaluation support, security, debugging support, and a highly scalable thread-safe network communication library.
- Support for shared objects on heterogeneous platforms. To improve performance and mitigate communication latencies, copies of objects can be created and accessed locally.
- Object sharing support that utilizes advanced distributed shared memory techniques for maintaining consistency of shared objects.
- The design of a "multiple protocol" approach which combines the capabilities of Mocha's network library and the TCP protocol to support efficient transfer of object replicas in a scalable fashion.
- Fault tolerance support that allows its overhead to be controlled based on the level of availability needed by an application for its objects.
- Empirical evaluation of the system in local area, wide area, and home service networks.

- Design strategies for developing visualization support for wide area computing.

In the next chapter, we present related work and its relationship to the wide area computing architecture we propose to investigate. Chapter 3 provides an overview of Mocha and its basic facilities. Chapter 4 introduces Mocha's shared object model, its basic algorithms and key implementation features, failure handling refinements, and provides an empirical evaluation of the system's state sharing capabilities. Chapter 5 presents an overview of applications developed for the framework. Chapter 6 discusses design strategies for developing visualization support for wide area computing and the concluding chapter discusses future work.

Chapter 2

Related Work

A substantial amount of wide area computing research currently ongoing is related to the focus of this thesis. In many cases, these systems leverage off of advances originally developed for distributed computing in local area networks. In this chapter, we provide an overview of several mechanisms for distributing an application developed for use in local area networks. We then provide an overview of several wide area computing environments and detail their approaches towards adapting traditional local area network mechanisms for wide area computing environments. A description of research areas that are peripherally related to this thesis work follows. Finally, this chapter concludes with a discussion of the goals and intended contributions of this thesis and its relationship to other wide area computing research currently under investigation.

2.1 Mechanisms for Distributing Applications

Several mechanisms for distributing applications in local area networks have been developed. Prominent examples include remote procedure call, remote evaluation, migration, message passing, and distributed shared memory. The following sections provide an overview of these mechanisms.

2.1.1 Remote Procedure Call / Remote Method Invocation

Remote Procedure Call (RPC)[8] is a widely used mechanism for building distributed applications. With this mechanism, a task calls a procedure that is located and executed on another machine. Message passing is utilized to transfer the appropriate parameters to the remote host as well as to return result values. The RPC model abstracts away the notion of performing message sends and receives and attempts to appear as similar to performing a standard procedure call as possible. Substantial effort has focused on making the remote procedure call transparent as well as improving the performance of the mechanism[60].

A variant of RPC is remote method invocation (RMI). RMI is essentially an adaptation of RPC for object-based distributed systems. With this approach, an object may invoke the methods of objects located on other hosts. In addition to the issues faced by RPC systems, RMI systems must provide facilities for the marshaling and unmarshaling of objects in order to permit them to be passed as parameters to a remote invocation.

2.1.2 Remote Evaluation / Migration

Remote evaluation[56] is a remote procedure call variant whereby the source or object code for the remote procedure is transmitted to the remote node responsible for executing the procedure. When large datasets are involved, it may be cost-effective to send the procedure to the dataset instead of sending data to the procedure. Remote evaluation has resurfaced as an important component of many wide area computing systems because in most cases the source or object code of a task is not assumed to be already installed at the remote location. The source or object code must be uploaded using remote evaluation techniques. The introduction of platform independent interpreted languages such as Java and Obliq has removed many of the impediments to heterogeneous remote evaluation such as incompatible binaries and data formats.

A facility related to remote evaluation is *migration*. With migration, a task is able to travel from one host to another, transporting itself and necessary data. Migration systems typically refer to their traveling tasks as *agents*. Bharat and Cardelli's Obliq based system[6] relies upon Obliq's distributed scope's semantics to support migration. Systems that provide migration support typically use a "suitcase" methodology whereby the agent explicitly states which data must be migrated with the task[6, 33]. This approach is much more efficient than blindly migrating a task's complete address space but requires more programmer effort than the latter technique.

2.1.3 Message Passing

Message passing is an established paradigm for developing high performance applications in network computing environments. Popular network computing environments that primarily utilize message passing as the mechanism for writing a distributed application include PVM[58] and MPI[47]. Both of these systems provide primitives for message sending and receiving, and related activities such as broadcasting and multicasting. Essentially, these systems provide a framework which emulates a distributed memory multiprocessor in networked computing environments composed of workstations, mainframes, and hardware multiprocessors. When developing a distributed application with the message passing model, a user explicitly programs activities such as performing message preparation, message transmission and reception, and message ordering. These activities can potentially increase the complexity of programming distributed applications.

2.1.4 Distributed Shared Memory

Distributed Shared Memory (DSM) is emerging as a popular abstraction for programming distributed systems. Typically, programming an application is simpler with shared memory than using the standard message passing model. Message passing programming becomes quite complex for applications that have complicated communication patterns. The shared memory model eases this complexity as it allows uniform access to both local and remote information through memory operations.

Many DSM systems are based upon the release consistency shared memory model. Release consistency was originally developed for use in the DASH shared memory multiprocessor [24]. With release consistency, synchronization operations are partitioned into two types, *acquires* and *releases*. This permits the DSM system to determine the difference between entering and leaving a critical region which can lead to a more efficient DSM implementation[60]. There are two popular implementations of release consistency, eager and lazy. In the eager release consistency model[24], all `STORE` accesses in a synchronization section are performed (i.e., made visible to all) before the subsequent *release* may be performed. In a distributed system environment, this typically requires a broadcast of all `STORE` access modifications (executed in the synchronization section) to all other tasks. In lazy release consistency[37], this information is only sent to the task that next executes an *acquire*. Hence, the propagation of modifications are further postponed until the time of the next *acquire*. Vector timestamps are utilized by the process that releases a lock to identify what new modifications must be propagated to the new process that acquires the lock.

Another consistency model which has been utilized to implement DSM systems is entry consistency[5]. The entry consistency model is similar to release consistency in that it also partitions synchronization operations into *acquires* and *releases*. Moreover, of the two forms of release consistency, entry consistency is most similar to lazy release consistency because `STORE` access modifications are only sent to the task that next executes an *acquire*. Entry consistency distinguishes itself from lazy release consistency by requiring that shared variables or objects must be explicitly associated with a lock. This requirement reduces the overhead associated with acquiring or releasing a lock because it can be quickly determined which shared variables or objects are associated with a lock and thus which variables or objects need updated. With release consistency, vector timestamps associated with the shared data must be examined to determine what variables or objects must be updated when an *acquire* is performed.

The development of DSM systems has been the focus of a substantial amount of research. More information on other significant DSM systems can be found in [34, 1, 18, 45, 51].

2.2 Wide Area Computing Systems

While the mechanisms for building distributed applications described in the previous section were intended primarily for use in local area networks, they are now being leveraged for the development of wide area computing systems. This section provides brief descriptions of several wide area computing systems currently under development.

The Legion system is an object oriented middleware developed for wide area computing[27]. Legion utilizes remote method invocation, the remote procedure call variant as its mechanism for distributing computation. With this approach, a Legion object may invoke the methods of objects located on other hosts. The Legion system then performs the necessary network communication to send method parameters and return the results of the method (if applicable). In addition, the Legion system provides a limited form of remote evaluation that allows the modification of server-side code[27]. An event driven model is supported whereby an arriving message contains a list of functions that must be called in order. With this model, the user is able to choose which routines are executed. This model is also flexible enough to implement active messages. The Legion system is not based on a platform independent language such as Java and therefore must mitigate complexities of heterogeneity without the support a platform independent language inherently provides.

Chandy's worldwide distributed system[12] is a Java based wide area computing system which utilizes inboxes and outboxes connected by message channels to enable distributed tasks to cooperate. With this technique, a process can append messages to the tail of one of its outboxes and remove messages from the head of its inboxes. Message channels connect an outbox to an arbitrary number of inboxes (and vice versa) and the channels deliver messages in the order they are sent. Although this approach is very similar to standard message passing, the channels have the distinguishing aspect of adding logical clocks that satisfy Chandy's global snapshot criterion. This criterion provides the system with the ability to perform distributed snapshots to determine global states. This facility is useful for determining stable properties of a distributed application such as application termination and detecting deadlocks[13]. The IceT[26] wide area computing system also utilizes message passing support for task cooperation, and it uses a model that is similar to that provided by popular message passing systems such as PVM.

Another Java based system is Atlas[3]. The Atlas system provides remote procedure calls (RPC) and permits these calls to be nested. This essentially allows Atlas to support subtasks that return results to other tasks. This communication model is somewhat limiting because it does not allow a task to communicate with any arbitrary task but only with subtasks that it has spawned. Nonetheless, this model is useful for many data parallel applications such as replicated concurrent simulation[59]. Another system that relies on RPC is NetSolve[11]. With NetSolve, the remote procedure task to be performed is executed as a parallel application by using an appropriate high performance technology such as PVM, MPI, HPF, or LINPACK.

ParaWeb is a Java based system[10] that modifies the Java interpreter to provide a global shared address space using distributed shared memory techniques pioneered by systems such as Munin and Treadmarks. In the ParaWeb implementation, the Java interpreters have been modified to permit them to cooperate and maintain the illusion of global shared memory. ParaWeb utilizes Java's built-in synchronization facilities to monitor when remote memory must be updated to maintain the illusion of consistent global memory. The ParaWeb system also provides a Java Parallel Class Library (JPCL) that supports standard message passing.

PageSpace[15] relies on a Linda-like coordination technology developed in Java as its mechanism for facilitating task cooperation. Essentially, PageSpace supports a global tuple space. Nodes may insert or remove tuples from this space without any regard for where the tuples are stored.

The TACOMA system relies on migration as it mechanism for supporting distributed computation and uses a suitcase methodology for transmitting shared state[33]. A major contribution of the TACOMA system is that it does not require that the TACOMA software has been installed at any host that might launch or be visited by an agent. Instead, TACOMA provides a method for instantiating agents on machines that have not installed TACOMA software. In this scheme, a user constructs an agent. A Web browser is then used to submit the agent (just as one might submit a survey form to Georgia Tech's World Wide Web Survey[49]) to a site that does have TACOMA software installed. This site instantiates the agent and returns its results to the original web browser in the form of a newly generated html page.

The Agent Tcl[41] system utilizes migratable agents that are an extension of the Tcl scripting language for its wide area computing activities. A novel aspect of the Agent Tcl system is its *laptop docking system* which allows agents to migrate to hosts that may become disconnected from the network. An agent that wishes to migrate from a disconnected host is queued until the host is reconnected. A *dock-master* is responsible for monitoring for reconnection and performs agent transfers immediately upon host reconnection. This approach is much more robust than the traditional timeout approach which has a higher potential for failure. With the traditional timeout approach, an agent may be sleeping during the period of reconnection and miss its opportunity to migrate.

Some researchers are exploring the synergy of high performance computing environments and Web technologies as a foundation for wide area computing. WebWork

System	RMI/RPC	Message Passing	State Sharing	Fault Tolerance
Legion	∘			∘
Chandy		•		
IceT		•		
Atlas	•			•
NetSolve	•			
ParaWeb		•	•	
PageSpace			•	
Tacoma				
Agent Tcl				
WebWork		•		

KEY: ∘ Some Support • Strong Support

Table 1: Summary of task communication and fault tolerance facilities provided by various wide area computing systems.

is one such system[21]. WebWork leverages off the public domain and commercial tools developed for the World Wide Web (WWW) to provide its wide area computing environment. The approach naturally links high performance computation with the collaboration capabilities of the WWW. WebWork relies on the WWW to support collaborative rapid prototyping of applications, distributed processing via HTTP message passing channels, and the spawning of computational modules through the CGI protocol. WebWork expects to continue to leverage off of the underlying Web software infrastructure and the explosion of popularity of Web software is expected to guarantee that this infrastructure will continually improve in quality and performance.

Table 1 presents a summary of task communication and fault tolerance facilities and Table 2 shows a summary of the miscellaneous mechanisms provided by the wide area computing systems discussed in this section.

System	Remote Evaluation	Migration	Security	Platform Independence
Legion			∘	
Chandy				•
IceT	•		•	•
Atlas	•			•
NetSolve				
ParaWeb				•
PageSpace				•
Tacoma		•		
Agent Tcl		•		•
WebWork	•			

KEY: ∘ Some Support • Strong Support

Table 2: Summary of miscellaneous facilities provided by various wide area computing systems.

2.3 Other Related Work

There are several facets of computer science research that are peripherally related to wide area computing, particularly in the context of this thesis work. Notable examples include fault tolerance, protocol support, security, visualization support, and computer supported cooperative work. The following sections discuss these research areas highlighting their relevance and use in wide area distributed computing environments.

2.3.1 Fault Tolerance

Because wide area computing involves a large number of geographically distributed hosts, there is a high probability that one or more hosts may fail during the execution of an application. Typically, fault tolerance support is provided in the form of a checkpoint and restart scheme. With this technique, a checkpoint is taken which stores enough information on stable storage to restart the application from the point of execution where the checkpoint was last performed.

The NetSolve[11] system uses a simple restart technique where the entire computation may be restarted should a failure occur. The Atlas[3] system provides more substantial fault tolerance support which relies on checkpoints and treats subcomputations as transactions. This strategy allows the side effects of a subcomputation to be visible as an atomic action which may be aborted. Currently, however, Atlas does not support shared variables between hosts and doing so in a fault tolerant manner would require enhancements to its current fault tolerance support.

The Legion[27] system allows applications to specify a level of fault tolerance and therefore applications which do not need fault tolerance do not have to incur the associated overheads. While Legion already provides fault tolerance for *stateless* objects, it is currently a high research priority to provide fault tolerance for *stateful* objects. The difficulty of stateful objects is an artifact of guaranteeing that the interaction of multiple objects results in each object seeing a consistent view of the world.

2.3.2 Protocol Support

Many wide area computing systems provide custom networking protocols to increase system performance and capabilities. The Legion system provides the Xpress Transfer Protocol. This protocol provides both the functionality of TCP and additional services such as reliable datagrams and priority messaging mechanisms especially designed for high-throughput, low latency communications. Legion's custom networking protocol allows the system to support not only file transfers and process to process communication, but also multimedia applications.

The I-WAY system[20] provides a software architecture referred to as *multimethod* *communication*. This architecture allows multiple communication methods (e.g., ATM Adaptation Layer 5, Myrinet, shared memory, TCP, UDP, IBM MPL, etc.) to be supported transparently in a single application. The method of communication may be chosen automatically or by user-specified selection criteria. The I-WAY system relies on a global pointer construct to maintain information regarding the methods available to perform communication with remote nodes.

Active IP[66] is a network architecture that allows normally passive IP packets to contain encapsulated program fragments. These program fragments may then be used by the network to perform customized computations on user data that travels through the network.

2.3.3 Security

Because remote evaluation and migration may result in untrusted code fragments executing on several hosts, a wide area computing system must be extremely concerned with security. At the very least, the system must guarantee that untrusted code is not able to corrupt filesystems or otherwise damage a remote host. Java based systems enjoy the luxury of Java's *SecurityManager* class which allows for comprehensive security policies for the execution of untrusted code fragments. Wide area computing systems not based on Java (or a language with similar security support) are at a distinct disadvantage when attempting to convince users that the system is secure.

Many non-Java systems simply limit capabilities to avoid security concerns or currently ignore addressing this issue. The NetSolve system does not support remote evaluation nor migration and as a result untrusted code is never executed. Hence the need for security is diminished. WebWork does not currently provide security policies for its remote evaluation capabilities nor does it discuss possible future implementations.

The Legion system provides security by allowing users to implement their own security policy or to use an existing policy via inheritance. For example, a user may choose a class that requires every method to have its parameters encrypted. Furthermore, as described above, Legion's remote evaluation capabilities are quite limited and this reduces the need for security.

2.3.4 Visualization Support

Understanding and analyzing the execution of a wide area distributed application can be a difficult task. One approach to addressing the complexities associated with distributed computing is the use of visualization. Visualization has been shown to be an effective aid for program understanding, debugging, and performance tuning. Wide area computing systems also stand to benefit from such support because wide area applications inherit the complexities common to all distributed environments.

There has been little work in the area of wide area computing visualization tools. Some groups are relying upon network computing visualization tools when possible. For example, the Wide Area Metacomputer Management (WAMM) system is a PVM based wide area computing tool under development at the University of Pisa. The developers of WAMM are currently investigating the integration of the PVaniM visualization environment into WAMM[4].

A fairly substantial amount of work has been done in the visualization of parallel and distributed systems[42]. Because wide area computing systems are closely related to traditional parallel and distributed systems, it is worthwhile to mention the recent uses of visualization for various aspects of parallel and distributed computing.

Perhaps the most common activity in parallel computing for which visualization has been used is performance evaluation[30, 29]. The ParaGraph[31] system is perhaps the best known application of visualization to parallel programming for this purpose. ParaGraph provides general purpose visualization support for performance analysis and tuning and utilizes the PICL[23] trace file format. In order to use ParaGraph, a parallel program must use the PICL communication facilities, or emulate their trace format. Many of the views illustrate the behavior of message passing activities performed by the distributed or parallel application. Other views used by ParaGraph include critical path views, Gantt charts, and Kiviat diagrams.

Visualization has also been used for the debugging of parallel and distributed applications. Some techniques rely on the examination of trace events to debug applications[46]. Other tools such as TraceViewer are able to detect and report race conditions[32]. Displays provided by TraceViewer include an event history graph, the source program, and a data race report.

Another use for visualization is program understanding. Frequently, the operation of a program and its algorithm or methodologies must be conveyed to individuals. Software visualization displays that illustrate the data structures, algorithms, and operations of a parallel program can be helpful to facilitate program understanding. The types of displays used for this purpose often must be application-specific[54, 57].

Visualization has also been used for monitoring the environment in which a parallel or distributed application is executing. An early system that provided graphical displays regarding environment was PIE[43]. The PIE system provided displays regarding context-switching patterns as well as the mapping of threads to processors, thus giving the user a feel for the environment in which their program executed.

2.3.5 CSCW

Computer-supported cooperative work, or CSCW, is an area of research that is concerned with using computers in a fashion that supports and enhances the work activities of groups[19]. Because these groups may involve individuals that are geographically distributed, CSCW can be viewed as a variant of wide area computing.

CSCW research predominately focuses on two main aspects. The first aspect is the social and cultural issues associated with collaborative systems. Examples of such issues include (i) addressing the need for privacy, (ii) providing for the peripheral awareness of others, (iii) facilitating the initial rendezvous or coming together of collaborators, and (iv) delegating control over the environment to collaborators[19].

In addition, a second aspect CSCW systems must address are architectural issues such as (i) supporting multiple input streams, (ii) providing serialization and synchronization support for multi-user input, and (iii) providing models of information sharing[19]. In some cases, CSCW systems provide their own solutions to the above architectural issues[19].

In many cases, however, CSCW research is complementary to the wide area computing research that is ongoing. Therefore, we expect CSCW applications to leverage off of wide area computing advances to better solve the architectural issues these systems face. For example, CSCW applications might leverage off of remote evaluation support provided by a wide area computing system to simplify the complexity associated with instantiating CSCW application tasks at remote sites. The robust heterogeneous shared object models being developed for wide area computing can be utilized by CSCW researchers to provide more useful models of information sharing. Finally, advanced techniques used to support synchronization can be used by CSCW systems to provide more efficient synchronization support for the serialization of multi-user input. Similarly, wide area computing systems can leverage off of implementing classical CSCW applications (e.g., shared calendar) to illustrate that system contributions investigated by these systems may be effectively utilized by real world applications.

2.4 Discussion

With regard to the wide area computing framework we investigate, many of the characteristics of wide area computing systems previously presented are related to the goals and objectives of this thesis work, namely, the development of system functionality that facilitates the programming of wide area applications that span the home as well as those utilized for metacomputing applications. We develop a system Mocha that supports the following mechanisms to achieve the goals and objectives of this thesis work:

- *Platform Independence*–By focusing our efforts on developing a system that is 100% pure Java we have the best opportunity of creating a framework capable of supporting heterogeneous environments such as those expected to be encountered in both home service applications as well metacomputing applications.
- *Remote Evaluation Support*–We anticipate remote evaluation support to be a critical component to our framework. We feel users will not be willing to constantly arrange for formal file transfers into guest accounts (or onto home

machines) in order to execute a task at a remote site. Furthermore, we feel techniques such as those used by Legion to allow messages to mandate which function is executed at the remote site are too restrictive for general purpose remote evaluation.

- *Protocol Support*–Similar to other systems, we feel custom network protocol support is needed to improve the performance and utility of the system. Off the shelf protocols for use on the Internet have disadvantages that limit their usefulness for wide area computing. For example, the Universal Datagram Protocol (UDP) is lightweight but unreliable. The Transmission Control Protocol (TCP) is reliable but heavyweight and tailored to bulk data transfer. In many situations wide area computing systems require network protocol support that is reliable, lightweight, and suitable for fine grain message passing between tasks.

- *Security*–Some form of security is also mandatory for our framework. Without it, no other sites will be willing to install the system. We rely upon Java's security support to implement a comprehensive security policy system. Not relying upon a Java based solution yet permitting remote evaluation of user application classes typically results in the following approach: relying on encryption for digital signatures to verify that untrusted code comes from a source that you *hope* will not send code that corrupts your system. In fact, this is the solution employed by Microsoft's ActiveX. Unfortunately, having to rely on the goodwill of others to send "safe" code to insure a secure system is an ad hoc solution at best and an invitation for disaster at worst.

- *State Sharing Support*–With respect to task communication models, we feel message passing is simply too primitive for use in a wide area distributed computing environment. Debugging message passing applications is already a complex activity in a local network environment. In a wide area environment, where users may not have accounts on the machines they are using, spawning a debugger to discover a bug related to improper message passing may not be an option. As an alternative to message passing, we investigate the development and efficient implementation of a robust shared object model as our task communication model that is capable of supporting general purpose shared objects. In contrast to the state sharing mechanisms investigated by other wide area computing systems such as ParaWeb[10], Legion[27], or PageSpace[15], the focus of our research contribution is the development of a shared object model for wide area computing that combines advanced distributed shared object techniques with failure detection and handling support that allows its overheads to be controlled based upon the level of availability needed for the shared objects.

- *Tools*–Additionally, this work investigates the development of tools that simplify the creation of a wide area computing environment and thus provides scaffolding that permits those with little or no computer science background to be able to start a wide area application.

- *Debugging and Visualization Support*–The development of modest debugging support and the design of a visualization subsystem are described. The purpose of this support is to enhance the usability and functionality of the wide area computing system and its ability to facilitate the development of applications that span the home as well as those utilized for metacomputing activities.

In the next chapter, we provide an overview of the Mocha framework, its basic facilities, and a description of its architectural philosophy and design.

Chapter 3

Mocha Overview

To motivate the need for a framework such as Mocha, we begin with a description of a potential wide area distributed application. Consider an application in which a consumer at home wishes to add a new formal dinner table place setting composed of flatware, plates, and glassware. At the consumer's home, a graphical user interface is executing that allows various flatware, plates, and glassware to be viewed together so that the consumer may "mix and match" these items and end up with a pleasing coordinated table setting. Additionally, a sales associate located at the retail outlet may also have a copy of the graphical user interface that permits the associate to see what the customer is selecting and may suggest alternatives which are then presented in the customer's GUI. Furthermore, the home consumer may have requested friends located at other homes to also participate in this decision making and therefore they too may be running a GUI and viewing the possibilities and also making suggestions. In this scenario, it is expected that the platforms on which the GUI executes in each of the homes would be vastly different from the platform at the retail outlet.

For a wide area computing infrastructure to support the above scenario, there are several features it must provide. First, it must provide support for shipping platform independent GUI code to the remote sites and allow the code to execute in a secure environment. Second, the infrastructure must provide task communication models that allow the GUI's at each of the sites to cooperatively work together. Finally, the infrastructure must enable applications to be written such that they are resilient to failures at the remote sites.

The Mocha system is capable of supporting an application scenario such as the one described above. It is written entirely in Java and provides the application programmer with a framework for developing wide area distributed applications in the Java language. In addition to providing constructs for distributed computing such as the remote spawning of threads and state sharing between these threads, the Mocha system provides features that enable it to serve as a wide-area computing environment. These include the ability to ship and dynamically link in application code (i.e., remote evaluation support[56]), network communication support, security, basic debugging and event logging facilities that provide insight into the execution of code at remote locations, and tools for initiating a session of wide area computing. The following sections provide a more in-depth overview of the facilities provided by Mocha.

3.1 Mocha's Architecture Philosophy

Mocha assumes that at each site (i.e., workstation) that intends to participate in the execution of Mocha applications, a *Site Manager* will be started that will listen on a well known port. The *Site Manager* listens on this port for requests to utilize the workstation and is responsible for controlling the number of true processes on the workstation that are allocated for use by remote tasks. These processes are referred to as *Mocha Servers* because they are able to "serve" a thread executing in the process with anything it may need. For example, a *Mocha Server* provides a thread with the ability to receive and link in more application code, as well as other features such as communicating results and reporting error conditions. Inside a *Mocha Server*, a thread may spawn a substantial number of threads. Obviously, spawning a substantial number of threads may seriously degrade the performance of the application. However, since these threads are isolated in a single process, tasks owned by other users of the workstation are relatively immune from the effects of this ill-advised activity because they run in separate *Mocha Servers* that are distinct processes. In modern operating systems such as Unix and Windows NT, processes are timesliced and therefore all *Mocha Servers* will receive a fair amount of cpu cycles. Furthermore, should a thread somehow find a way to crash a whole *Mocha Server*, threads belonging to other wide area computing applications are protected from this catastrophe. In addition, the number of Mocha Servers permitted to execute simultaneously can be controlled by the *Site Manager*.

3.1.1 Task Communication Models

Wide area computing applications rely on some form of communication model to permit tasks to work cooperatively. The Mocha system provides two task communication models. These include a travel bag based task communication model and a state sharing model based upon robust shared objects.

In the travel bag based task communication model, a task that is instantiated on a remote host is provided a travel bag. This travel bag provides the task with the ability to retrieve initial execution parameters, return results, and provides an interface to allow the task access to capabilities such as remote printing and performing remote stack output. This model is expected to support basic wide area computing applications such as data parallel applications, Internet surveys, and Internet advertisements. Additionally, this model's communication facilities are used by Mocha's more general state sharing model for initialization purposes. A more in-depth description of the travel bag task communication model is provided in Section 3.2 of this Chapter.

Mocha's state sharing task communication model allows for more general communication between all tasks by providing a state sharing scheme whereby updates at one replica are reflected at all other replicas. Mocha's state sharing utilizes efficient mechanisms for the consistency maintenance of shared state across Mocha servers running at different nodes. It also supports the sharing of complex objects and provides failure handling support that allows its overhead to be controlled based on the level of availability needed by an application for its objects. Mocha's state sharing model is the focus of Chapter 4.

3.2 Remote Evaluation

In the Mocha environment, the application is composed of threads that execute in the address spaces of Mocha Servers. These threads may be initiated (i.e., spawned) using a method from the provided `Mocha` class. A typical section of code executed by a thread to spawn another Mocha thread is shown in Figure 1. When a new instance of the Mocha object is created, a host file is read that provides a list of potential

```
import mocha.*;
public class TestMocha  {

   public static void main(String args[]) {
      Mocha mocha;
      ResultHandle rh;
      Parameter p;

      mocha = new Mocha();
      p = new Parameter();
      p.add("param1", 5);       // create parameters to
                                // send to remotely evaluated class

      rh = mocha.spawn("Myhello", p); // spawn class named Myhello
                                      // remote site requests other
                                      // classes as necessary

   }
}
```

Figure 1: Mocha application code that spawns a class for remote evaluation.

sites at which remote threads may be spawned. The Mocha system provides a tool to generate this host file. As shown in the code fragment, a `Parameter` object is utilized for organizing the parameters that will eventually be sent to a remotely spawned thread. The actual spawn is performed by calling the `spawn()` method of class `Mocha` with the name of the Mocha thread class that is to be spawned and the `Parameter` object that is to be sent to the remotely instantiated thread. Other spawn methods are available that allow the application to specify the exact host in the host file on which a remote thread should execute. In Figure 1, the class being spawned as a thread is the "Myhello" class. This and other classes that are intended to be spawned are provided by the application programmer.

In contrast to a spawn performed by a network computing environment such as PVM, Mocha's spawn provides remote evaluation support[56] that allows it to transport and dynamically link in thread code at a remote Mocha server as necessary. Mocha's model for remote evaluation is that of an initial "push" of application code followed by "demand pulling" of new application code object classes as they are encountered during execution. The Mocha system relies upon its own communication substrate to perform this transmission.

Mocha threads may be derived from any Java class that implements the `MochaTask` interface. Through this interface the Mocha runtime is able to provide the application thread with a `Mocha` object that is essentially a "travel bag". The `Mocha` object currently provides a `Parameter` object from which the remotely evaluated task may retrieve the initial execution parameters denoted in the `spawn()` method that started this task. Also provided is a `Result` object in which the task may place results. Additionally, the `Mocha` object provides a variety of useful objects and methods that support remote printing, remote stack dumps, support for making replicated object copies and accessing the replicas, and enables a thread to recursively spawn other wide area computing threads.

Figure 2 shows an example user class that utilizes the `Mocha` class to acquire initial startup parameters, perform remote printing and stack dumps, and then return its subresult. In this example, `Myhello` is a user class that has been written in a fashion that permits it to be shipped and executed at a remote site. After the class is shipped, the Mocha runtime creates a new thread that begins executing at the `mochastart()` method. The Mocha runtime provides this thread with a `Mocha` object from which the thread may invoke the travel bag related activities described above. Figure 3 shows an example user class that utilizes the `Mocha` class to spawn tasks to compute π in parallel. As illustrated in the Figure, this application spawns several threads

```
import mocha.*;
import java.lang.*;

public class Myhello implements MochaTask {
  public Myhello() {
  }
  public void mochastart(Mocha mocha) {
    double start=0.0, sum = 0.0;
    try {
      start = mocha.parameter.getdouble("start");
      sum = start + 1;

      mocha.mochaPrintln ("Returning as a return value " + sum);
      mocha.result.add("returnvalue", sum);
      mocha.returnResults();

    } catch (MochaParameterException e) {
      mocha.mochaPrintStackTrace(e);
    } catch (Throwable t) {
      mocha.mochaPrintStackTrace(t);
    }
  }
}
```

Figure 2: Remotely evaluated user class that illustrates the use of the Mocha object.

which begin executing in the class `PiTask`. Each thread is provided with startup parameters that enable the thread to determine which portion of the computation the thread is responsible for. The thread then waits to receive the partial results using the `waitForResult()` method and then combines the results.

The corresponding `PiTask` application code is shown in Figure 4. As illustrated in the Figure, each thread first acquires initial startup parameters, computes a portion of π, and then returns its subresult.

3.2.1 Design

The Mocha system relies heavily on Java's support for dynamic class loading to support remote evaluation. Java provides a `ClassLoader` class which Mocha subclasses as a `NetworkClassLoader` to define its own policy for dynamic class loading and instantiation. The Java runtime relies upon Mocha's `NetworkClassLoader` to load any class that has any relation to a dynamically loaded class. Thus, if the Java runtime encounters a class that needs to be dynamically loaded or if this class is being referenced by a class that has already been dynamically loaded, the Java runtime will delegate the process of dynamic loading to Mocha's `NetworkClassLoader`. This proceeds in a manner analogous to an upcall[60] whereby the Java runtime calls a method of Mocha's `NetworkClassLoader` passing in a parameter that is the name of the class to be dynamically loaded. Mocha's `NetworkClassLoader` then performs the necessary network communication to acquire the class and load it locally into the Java runtime. Mocha's model for remote evaluation is that of an initial "push" of application code followed by "demand pulling" of new application code object classes as

```
import java.lang.*;
import mocha.*;
public class PiMaster  {

  public static void main(String args[]) {
    Mocha mocha;
    ResultHandle rh[];
    Result result;
    int value;
    Parameter p;
    double tmp=0;
    double start= 0.0;
    double delta = 0.000001;
    int ntasks = 4;
    rh = new ResultHandle[ntasks];
    mocha = new Mocha();
    double sum = 0.0;
    for (int i=0; i< ntasks; i++) {
      p = new Parameter();
      p.add("start", start);
      p.add("delta", delta);
      p.add("ntasks", ntasks);

      rh[i] = mocha.spawn("PiTask", p);
      start = start + 1.0/ntasks;
    }

    for (int i = 0; i < ntasks; i++) {
      result = rh[i].waitForResult();
      try {
        tmp = result.getdouble("returnvalue");
      } catch (MochaParameterException e) {e.printStackTrace();}
      sum += tmp;
    }
    sum = sum*4.0;
    mocha.mochaPrintln("Pi =  " + sum);
  }
}
```

Figure 3: Mocha application code for the root task of parallel π computation.

```
import mocha.*;
import java.lang.*;

public class PiTask implements MochaTask {
  public PiTask() {
  }
  public void mochastart(Mocha mocha) {
    double start, delta, sum=0.0;
    double end, x;
    int ntasks;
    try {
      start = mocha.parameter.getdouble("start");
      delta = mocha.parameter.getdouble("delta");
      ntasks= mocha.parameter.getint("ntasks");

      end = start + 1.0/ntasks;

      for (x=start; x < end; x+=delta)
         sum += Math.sqrt(1.0 - (x*x)) * delta;

      mocha.mochaPrintln ("Returning as a return value " + sum);
      mocha.result.add("returnvalue", sum);
      mocha.returnResults();

    } catch (MochaParameterException e) {
      mocha.mochaPrintStackTrace(e);
    } catch (Throwable t) {
      mocha.mochaPrintStackTrace(t);
    }
  }
}
```

Figure 4: Mocha application code for a worker task of parallel π computation.

they are encountered during execution. The Mocha system relies upon its own communication substrate to perform this transmission and this subsystem is described in the next section.

Although Java's support for a custom, dynamic `ClassLoader` is powerful, it places substantial responsibility on the implementor of the `ClassLoader` (i.e., Mocha) to insure an efficient implementation. Mocha increases the efficiency of remote evaluation by relying on the following optimizations to its custom `ClassLoader`:

- *Caching of dynamically network-loaded classes* –Whenever the Java runtime encounters a class that requires dynamic loading, it always relies on the custom network loader to load the class even if the it has already previously loaded the class. Thus a naive custom loader could potentially request the same class over the network multiple times for different instances of a particular class. The Mocha `NetworkClassLoader` avoids this redundant use of network bandwidth by caching classes once they have been remotely fetched. Mocha caches classes locally using a hash table to insure efficient lookup.

- *Short-circuiting dynamic network-loading when possible*–Whenever a dynamically loaded class references another class, the Java runtime system directly relegates the task of loading this class to Mocha's `NetworkClassLoader`. While in some cases this may be a user class that must be fetched, often the class referenced is a portion of the Mocha library or the standard Java library. These libraries are already present at all sites and remotely fetching classes from these libraries is a costly, unnecessary action. Mocha avoids this unnecessary remote fetch by recognizing these standard classes and dynamically loading them from the locally available copies.

The process of network loading is initiated through the use of a `spawn()` method. Pseudo code for this method is provided in Figure 5. This method enables an application thread to spawn a thread for execution in a *Mocha Server* on a selected remote host. A `Parameter` object may be passed in to the method that contains parameters that need to be shipped to the remotely instantiated thread. If a host has not been explicitly chosen by the thread invoking this thread, a default host is selected. The method then creates textual displays if this spawn is being performed on the root host and the displays have not already been created via a previous `spawn()` method call. A surrogate thread is then created and is responsible for providing the remotely instantiated thread with class files that it may request. A request message is sent to determine if it is permissible to execute the thread in the remote `MochaServer`. If the request is granted, a new thread is created at the remote site and is provided the initial startup parameters by placing a copy of the `Parameter` object in its travel bag object.

A `NetworkClassLoader` object is bound to a thread by loading the initial class to be remotely executed using a `loadClass()` method of `NetworkClassLoader`. The binding is completed by having the newly created thread invoke a `mochastart()` method that exists in the class file that is initially transferred for remote execution. After the binding is complete, the executing thread will invoke the `NetworkClassLoader` whenever it creates a new instance of some object or must execute a method of another object. At this point the `NetworkClassLoader` determines if the class file is already locally available from a previous request, having already been transferred over or is a part of the standard Java or Mocha libraries.

```
public ResultHandle spawn (String classname, Parameter param, int where) {

   if (no target host has been selected)
       choosehost();     // choose host in a roundrobin fashion

   if (textual output displays need to be created)
       createTextualDisplays();

   createSurrogateThread();  // create local surrogate thread to provide
                              // class files and perform print duties
                              // for remotely spawned tasks

    msg = new MochaMsg();
    msg.packUTF(surhostname);  // pack address and port of surrogate
    msg.packInt(surport);

    rh = createResultHandle();
    packHostList();
    packParameters();

    if (requestHostUse()) {
        mochaLog("spawning class " + classname +
                 " on " + hostobj.hostname);
        return(rh);
    }
    else {
        mochaLog("Error: cannot spawn class " + classname +
                 " on " + hostobj.hostname);
        return(null);
    }


}
```

Figure 5: Mocha's implementation of its spawn method.

```
while(true) {

   msg = transport.irecv();

   if (msg.msgtype == REQUESTFILE) {
       if (class file exists locally) {
           send class file to requester;
       }
       else {
           forward request to my surrogate thread;
       }

   }
   else if (msg.msgtype == RETURNRESULT) {
            unpack results;
            create result object;
            place in result queue;
   }
   else if (msg.msgtype == PRINTMESSAGE ||
            msg.msgtype == LOGSMESSAGE  ||
            msg.msgtype == STACKDUMP)  {
            forward message directly to root host for printing;
   }

}
```

Figure 6: Mocha's implementation of surrogate threads.

If the class file does not fall into one of the above categories it must be requested from the surrogate thread located at the site that remotely spawned this thread. The surrogate thread is created specifically to satisfy the needs of the remote thread such as shipping its class files. Pseudo code for the operation of the surrogate thread is provided in Figure 6.

As illustrated in the Figure, the surrogate thread responds to a variety of requests made upon it by the remote thread. If the remote thread requires a class file to be shipped the surrogate thread determines if the class file exists locally. If locally available, the class file is sent to the requesting remote thread.

It is also possible that the class file is not locally available. This can occur because the Mocha system allows spawns to be nested. Thus, one thread could remotely spawn a thread that then in turn remotely spawns a thread. In this scenario, the last thread to be spawned may require a class file that does not exist on the host that spawned the thread but instead is located on the host that performed the initial spawn. In this case, it is necessary for the surrogate thread to forward the request to the surrogate that was created for its remotely spawned threads. This behavior is depicted in Figure 6.

The surrogate threads also respond to remote thread requests for printing, receiving stack dumps, and logging activities. These requests are made as part of Mocha's debugging infrastructure and this aspect is more thoroughly described in Section 3.5 of this Chapter. For these activities, the surrogate forwards the requests directly to the root host to reduce the latency of processing these requests.

3.3 Network Communication Support

Currently, the network communication substrates that are provided as part of the standard Java language include support for the Transport Control Protocol (TCP) and User Datagram Protocol (UDP) Internet protocols. The TCP protocol is a reliable connection-oriented protocol suitable for transmitting large byte streams. TCP, however, is not well suited for transmitting small messages because its connection and tear-down costs become significant overheads. The UDP protocol is lightweight but unfortunately unreliable. For network communication activities that require the reliable transmission of small messages, Mocha provides its own custom communication substrate. This substrate is utilized to support the remote fetching of class files, passing parameters, returning results, and other needs. Mocha's network communication library (NCL) leverages heavily off of Java's support for objects and threads to provide daemonless, lightweight communication protocol support.

Mocha's NCL is built directly upon Java's interface to UDP sockets. This connectionless protocol supports datagram messages which are unreliable, unsequenced, and have a maximum size of 65 kilobytes. Mocha's NCL implements reliability, sequencing, and performs fragmentation and reassembly to support messages of sizes greater than 65 kilobytes.

Mocha's NCL provides functionality that is in many ways similar to the communication substrate provided by the Parallel Virtual Machine (PVM)[58]. However, the implementation of Mocha's NCL is drastically different from PVM's communication substrate. PVM relies on a *pvmd*, a heavyweight process, that is responsible for receiving a message, sending acknowledgements, and performing reassembly. The message is then transferred to the user process using some form of interprocess communication, typically a Unix domain socket or pipe.

In contrast to PVM, Mocha's NCL is implemented completely in the user's process address space. Java's support for priority threads is utilized to implement responsibilities such as reassembly and acknowledgement sending that in PVM must be performed by a separate process, namely the *pvmd*. Moreover, Mocha's NCL is encapsulated into a `MochaXfer` class. This class provides methods analogous to PVM's message send and receive routines. The encapsulation of these activities into a class has resulted in a communication substrate that is cognizant of and amiable towards multiple threads of execution. In the Mocha system, different threads may use different instances of the `MochaXfer` class and hence may concurrently perform communication. In contrast, PVM was originally designed before the use of multiple threads of execution had become commonplace. As a result, the message passing routines were implemented as individual procedures that were not thread safe.

Mocha's NCL is scalable in the number of hosts that communicate with the library because it performs its own upward multiplexing of packets. That is, a thread can use a single instance of the `MochaXfer` object to receive messages from multiple distinct sites simultaneously. The NCL library uses a hash table to keep track of the sequence numbers from all the other `MochaXfer` objects being used by other threads sending messages. Message fragments from different threads are reassembled into messages, while insuring that fragments sent from different threads are not mistakenly reassembled into a bogus message. A `MochaXfer` object is lightweight because it only requires a single UDP port to be bound to the object.

Mocha's NCL is particularly well suited for sending small messages such as those used for debugging (described later in this Chapter), various control messages which include parameter passing and requesting class files from surrogate threads. This is because it avoids the heavy connection and tear-down overheads associated with other transport protocols such as TCP. These overheads would occur because TCP sockets have high system resource requirements which are hoarded by the TCP socket. This severely restricts the number of TCP sockets that may stay connected simultaneously. These resource requirements include *mbufs*, a type of memory buffer that the Unix operating system only provides a fixed amount of for all processes to use[7]. Thus, to use TCP in a scalable fashion for these fine grain message passing activities, its heavy connection and tear-down overheads must be continuously incurred. Empirically, we have found Mocha's network communication library to be approximately twice as fast as TCP for sending small (i.e., less than 256 byte) messages. For larger messages such as those that would be used to transfer large objects, we have found TCP's performance to be better than Mocha. A more in-depth evaluation of the network communication library's limitations and strategies for mitigating these limitations is presented in Chapter 4.

3.4 Security

The Java programming language provides a robust security model which allows systems such as Mocha to implement their own custom security policies. Essentially, Java permits Mocha to subclass from the Java `SecurityManager` class. This class provides methods that are associated with a variety of "precarious" actions. For example, consider the precarious action of a Java application attempting to write a file. Whenever the Java runtime encounters this type of code, it calls the `checkWrite()` method associated with Mocha's subclass of Java's `SecurityManager` class. At this point, the Mocha system may decide whether or not this action should proceed. Should Mocha consider the activity to be a possible security breach, it throws a security exception which prohibits the application from proceeding with this activity.

The Mocha security policy is somewhat complex because the Mocha runtime system requires capabilities such as reading and writing files, creating network connections, and dynamically loading libraries but must also forbid these activities to general purpose application code that is executing remotely via remote evaluation. Mocha's security approach relies on the following:

- *Recognizing an insecure action is being performed by the Mocha library*–Mocha's security manager examines the execution stack using methods subclassed from the standard Java class `SecurityManager` to determine if the potentially harmful activity is being performed by a method in the Mocha library. If this activity is being performed by a method in the Mocha library, it is assumed not to be a security breach. The activity is permitted to proceed.

- *Recognizing an insecure action may be performed by user code*–When application code is sent for remote execution via remote evaluation, it executes inside of a custom Mocha thread class aptly named `MochaThread`. Mocha is therefore able to determine if a precarious activity has the potential of resulting from unsecure application code and can throw a security exception. An example use of this capability is to guarantee that application code can never create a custom class loader. Custom class loaders have recently been used to breach security in the Netscape browser.

Figure 7 illustrates Mocha's security policy implementation for the insecure situation whereby application code is attempting to write a file. As shown in the Figure, the file modification is only allowed to proceed if one of the Mocha classes which performs file writes is on the execution stack. Otherwise, a security exception is thrown and the operation is aborted. Mocha's security manager provides protection from a substantial number of insecure activities and appears to be adequate. Very few security systems are one hundred percent foolproof and Mocha's security manager is no exception. It is anticipated however that many security holes can be identified and

```java
private boolean accessOK() {
  if (inClass("mocha.MochaXfer"))
    return true;
  else if (inClass("mocha.Mocha"))
    return true;
  else if (inClass("mocha.NetworkClassLoader"))
    return true;
  else if (inClass("mocha.FileCacheServer"))
    return true;
  else
    return false;
}

public void checkWrite(FileDescriptor fd) {
  if (!accessOK()) throw new SecurityException();
}
```

Figure 7: Mocha's implementation for secure file modification.

patched as a system such as Mocha receives more extensive use (as is the case with other tools that provide security such as Netscape's Java capable web browsers).

3.5 Debugging Support

Developing a wide area computing application is a very difficult endeavor. Tasks are executing on hosts on which users may not even have a guest login account. This, coupled with the added complexity that results from the communication and synchronization performed in a distributed application by individual tasks, may result in an environment in which it is extremely difficult to develop a bug free application.

Bugs in wide area computing programs may occur in any of the application's several components. A wide area computing application begins with the instantiation of threads at remote location and the passing of startup parameters. The initial spawn of a thread may be performed incorrectly or a parameter assumed to be accessible may not exist. After the initial startup period, wide area computing applications commonly access shared state variables, typically referred to as *replicas*. These replicas may erroneously be accessed before proper initialization. Furthermore, each task is at least a single thread of execution so all catastrophic bugs common in sequential programs (e.g. accessing an array out of bounds, dereferencing a null reference) are all possible in a wide area application. Finally, failing to comprehend the potential interactions between concurrently executing threads may also lead to bugs.

Since there are several components of a wide area computing application in which problems can appear, debugging support has the potential to greatly enhance the usability of a wide area computing system. Debugging support can provide the ability to report the values of variables used by remote threads. Debugging support can also provide information regarding the point of occurrence of catastrophic bugs such as those described above. Implementation of debugging facilities may include simple techniques such as allowing the system to perform remote print statements. Debugging may also encompass more complex techniques such as providing point of failure information by displaying a call graph up to the point of where a catastrophic bug occurred.

Mocha provides a modest set of debugging capabilities that include redirecting output, event logging of Mocha runtime activities, and remote stack printing. Figure 8 illustrates Mocha's capabilities for displaying output from tasks that are executing remotely. As shown in the upper text area of the Figure, the tasks are able to report partial result values to the central Mocha Task Output window. Tasks are identified by the host on which they are executing and by the host of their parent, which is shown in parentheses. The Task Output window also can be used to perform rudimentary

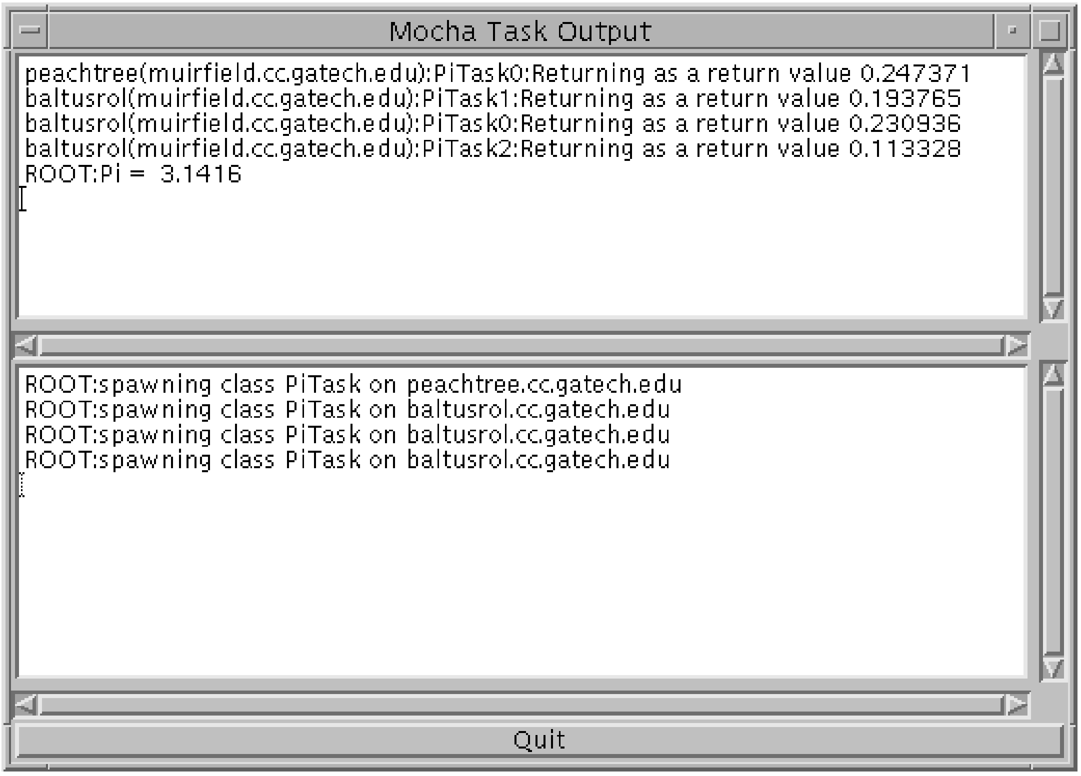

Figure 8: Mocha's Task Output display.

debugging of the executing tasks. Because Java debuggers are still immature, many Java programmers rely on adding print statements to their applications to perform debugging. The Mocha system provides this capability and therefore provides Java programmers with a method of debugging with which they are very familiar.

Mocha's remote task output capabilities are implemented using Mocha's own network communication library. A surrogate thread is provided on the root host to receive output messages, perform formatting, and update the Task Output window. One critical assumption made by our proposed framework is that the root host is considered to be more failure resilient than any other host in the wide area computing environment. The justification for this presumption is that the root host is the one workstation that is local, and the application developer has more control over this workstation than hosts that are remotely located. Thus, the root host is expected to encounter less unplanned program termination than tasks that run remotely "at will" of other users and their applications. For this reason, Mocha's critical features such as recording task output are performed at the root host.

The Mocha system also provides a log of activities performed by the runtime and places this information in a secondary window of the Mocha Task Output display. This window is shown as the bottom window in Figure 8. The log support is also implemented with the Mocha network class library. The textual log provides the user with insight into the online execution of the Mocha runtime.

Mocha's most powerful support for aiding users in debugging wide area applications is the ability to report to the root host point of failure information regarding catastrophic program errors that occur at remote sites even if the user does not have an account on the remote hosts. Mocha is able to provide a full call stack listing and the line number of where the remote program bug is located. Figure 9 illustrates Mocha's ability to provide remote stack output for two common program bugs. In the first stack listing, the application has used a Mocha library routine improperly. The task has attempted to retrieve a parameter as type `float` but Mocha realizes this parameter is actually an `int`. In this scenario Mocha reports to the root display that a type mismatch has occurred and provides a full callgraph listing that pinpoints where the error occurred.

In the second stack listing, a more dubious error has occurred. The application code has attempted to index an array beyond the bounds of the last element in the array. Again, Mocha provides a full callgraph listing and the exact location of the

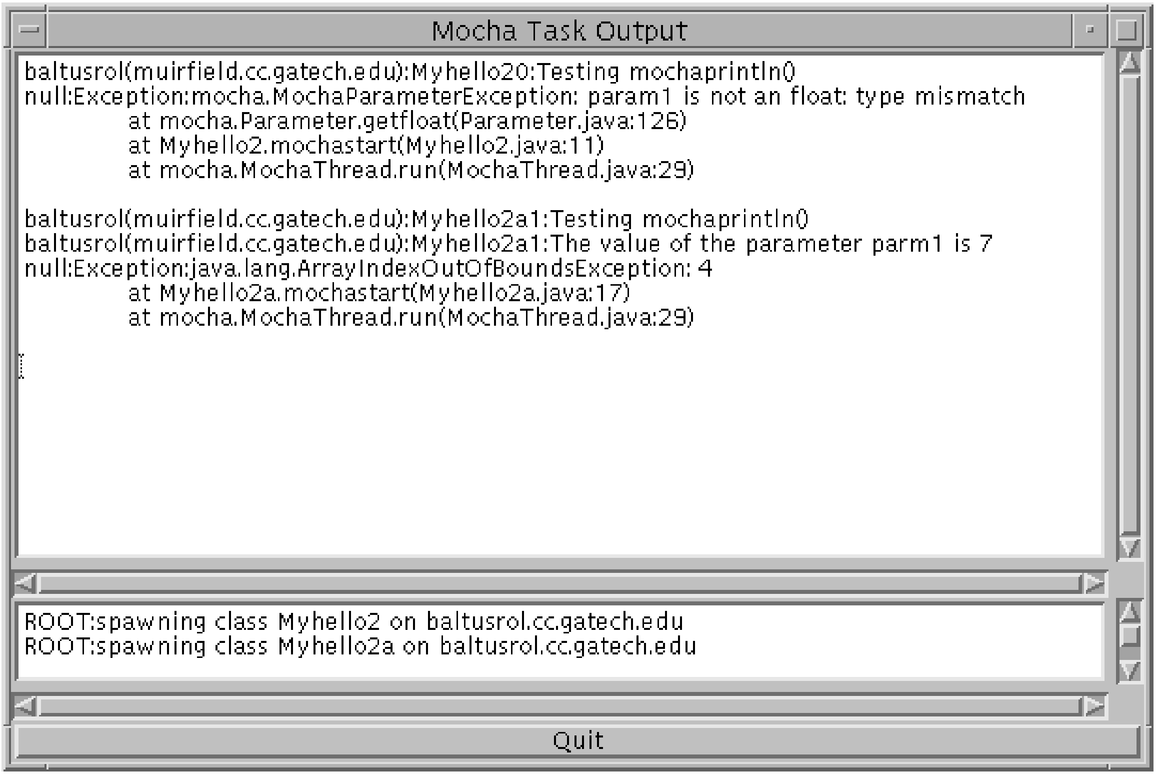

Figure 9: Mocha's remote stack output for task bugs.

error. Mocha also reports any additional information it has regarding the error. In this case it reports that the error is in fact an array index out of bounds exception.

3.5.1 Design

The design of Mocha's remote printing and logging features are relatively straightforward due to facilities provided by Mocha's network communication support and the existence of surrogate threads to receive remote printing and logging messages. The network communication library provided by Mocha is particularly apropos for sending small sized messages and most printing and logging messages fit into this category. Additionally, a surrogate thread exists on the root host to receive remote printing and logging messages and uses them to update textual displays.

Figure 10 provides pseudocode for the `mochaPrintln(String s)` method which implements remote printing. First, it is determined if the method is being invoked by a thread that has been spawned. If this is not the case, then the thread is executing on the root host and can directly access the appropriate textual display (i.e., printing or logging) and update the display. If the thread invoking the method has been spawned then it is aware of a surrogate thread on the root host to which it may send the string it wishes to remotely print or log. As illustrated in the pseudocode, in this situation the string is marshaled into a `MochaMsg` and sent to the surrogate thread on the root host.

Supporting remote stack trace printing is more difficult because when an exception occurs, some mechanism must exist to capture the stack trace which by default is sent to the screen. Mocha's solution to this problem is to exploit Java's stream model that decouples the consuming, processing, and producing of data from the sources and destination of that data[44]. This enables Mocha to redirect stack trace output to a byte array that can then be marshaled and sent to the root host for display. Figure 11 provides pseudocode for the `mochaPrintStackTrace()` method that may be used at remote hosts to perform remote stack tracing. The parameter to the method is an object of type `Throwable` which is the standard Java superclass for all exceptions that can occur during the execution of a Java application and thus enables the `mochaPrintStackTrace()` method to capture a stack trace no matter which exception may occur. The Java runtime passes exception information to an application via the use of its `try` and `catch` language construct. The `mochaPrintStackTrace()` method may be placed directly in a `catch` construct as illustrated in Figure 2.

```
public synchronized void mochaPrintln(String s) {

     ...

     if (!spawned) {  // if not a spawned task
         // update textual display
         mochaframe.printwindow.appendText("ROOT:" + s + "\n");
     }
     else {

       MochaMsg msg = new MochaMsg();  // create Mocha message

       // pack print message but first prepend task name
       msg.packUTF(Thread.currentThread().getName() + ":" +s);


       // send to root surrogate thread for processing
       transport.isend(rootsurhostname,
                       rootsurport,
                       msg,
                       transport.PRINTLN,
                       transport.SYSFLAG);

    }

     ...
```

Figure 10: Mocha's implementation for remote printing.

When a `catch` clause that contains a `mochaPrintStackTrace()` is invoked, the Mocha libraries can begin the process of shipping the track trace. As illustrated in Figure 11, a `ByteArrayOutputStream` is created and specified as the destination stream of a standard Java `PrintWriter`. This `PrintWriter` may then serve as the destination of stack trace output by calling the `printStackTrace` method of class `Throwable` and passing in the `PrintWriter` object to this method. At this point, the byte array associated with the `ByteArrayOutputStream` contains the captured stack trace. This byte array is then prepended with the thread identifier, marshaled into a `MochaMsg`, and sent to the root surrogate thread for processing.

3.5.2 Discussion

The debugging facilities provided by Mocha are admittedly limited compared to vast amount research that exists for parallel and distributed debugging. The major contribution of Mocha's debugging support is to automatically ship debugging information from a remote site back to the root site at which the application began execution. The implication of this for home service applications is that when catastrophic errors occur in the home, troubleshooting can in many cases be performed without it being necessary for a home user to communicate to a customer service center the exact errors that have occurred. The ability to avoid the need of feedback from the home user is critical since many home users are not expected to be comfortable with the activity of submitting bug reports.

```
public synchronized void mochaPrintStackTrace (Throwable t) {

    ...

    // create a ByteArrayOutputStream
    ByteArrayOutputStream bos = new ByteArrayOutputStream();

    // specify the ByteArrayOutputStream as the destination
    // stream of the PrintWriter stream
    PrintWriter ps = new PrintWriter(bos,true);


    // call method for printing the stack trace
    //  associated with an exception specifying
    // the PrintWriter stream as the destination
    t.printStackTrace(ps);

    // convert ByteArrayOutputStream to a String
    String temp = bos.toString();


    MochaMsg msg = new MochaMsg(); // create a Mocha message

    msg.packUTF(taskname + ":Exception:" + temp);  // marshal message

    // send to root surrogate  thread for processing
    transport.isend(rootsurhostname,
                    rootsurport,
                    msg,
                    transport.PRINTLN,
                    transport.SYSFLAG);


    ...
```

Figure 11: Mocha's implementation for remote stack printing.

3.6 Tools Provided By Mocha

Wide area computing environments rely on the ability to utilize machines separated by geographic distances. Furthermore, it is anticipated that the user may have permission to use a remote workstation but not have an account on the host. Wide area computing environments need to provide support to greatly simplify the laborious bookkeeping operations of creating a wide area computing environment. Moreover, these environments must provide support that allows users to spawn (i.e., start) a wide area computing application.

Without some form of tool support, creating a host pool for wide area computing is a burdensome task for users without a strong distributed computing background. The user would first have to find a way to contact a remote sight to determine that the site is online, most probably using a protocol such as `finger` or `ping`. Also, the user would probably rely on anonymous `FTP` to acquire information regarding host downtime schedules, rules regarding what and how many resources an application may use, and load average information for the remote host. After determining a list of feasible sites, the user must now pass this list to the wide area computing system (probably as a configuration file) in order to provide the runtime with possible sites on which it may spawn tasks. Finally, the user must begin executing the initial task that spawns remote tasks. In the case of an interpretive language such as Java, the user must also startup the Java interpreter.

For users at ease with distributed computing, manually performing the above steps is simply annoying. For users with less experience, the steps are serious roadblocks to developing wide area computing applications. We provide a user interface that

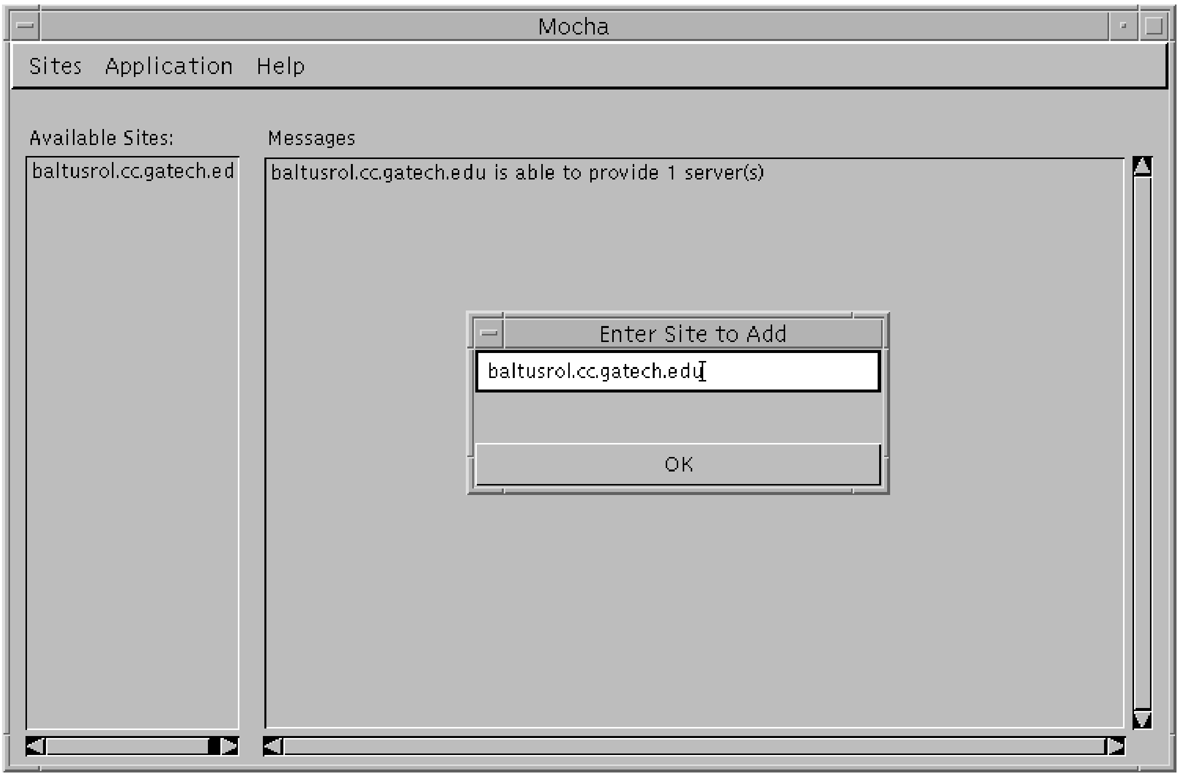

Figure 12: Mocha's user interface.

automates this process. This tool is shown in Figure 12. At the top of the interface is a series of pulldown menus that allow the user to perform various administrative activities such as add new sites and spawn applications. On the left side of the interface is a text area which displays sites that have been contacted and are now available for use. On the right side a text area is provided which provides messages to the user and output produced by the root task. As shown in the snapshot, the user has selected the *Add Site* option from one of the pulldown menus and the Mocha user interface displays a dialog box for the user to enter the host to use in the wide area computing session. The Mocha system then attempts to contact the site. Mocha assumes a *Site Manager* is executing at the site and is listening on a well known port.

The *Site Manager* decides whether or not it is willing to allocate *Mocha Servers* which as mentioned above are processes in which threads representing remote evaluations may execute. If the Site Manager is willing to allocate a *Mocha Server*, it responds to the request with the port number of the *Mocha Server*. At this point, a Mocha application that spawns tasks may directly interact with *Mocha Servers* allowing the *Site Manager* to now concentrate on requests by other users.

Spawning an initial Mocha application is also easily performed through the Mocha user interface. As shown in Figure 13, a file selection dialogue box is shown by Mocha's user interface when the spawn option is chosen from the `Application` pulldown menu. The user may then select a Mocha application to spawn. Mocha then instantiates the application and redirects its output to the Mocha user interface.

Mocha's user interface illustrates that it is possible to develop a platform independent tool that partially automates the construction of host pools for wide area computing sessions. The ability to abstract away the complexities associated with initiating a session of wide area computing eases the burden this procedure places on the user.

3.7 Conclusions

In this Chapter, we have presented an overview of Mocha's basic facilities which enable it to serve as a framework for wide area computing. These functionalities include the following:

- *Remote Evaluation*–Mocha's remote evaluation support permits the runtime to transport and dynamically link in thread code at remote *Mocha Servers*

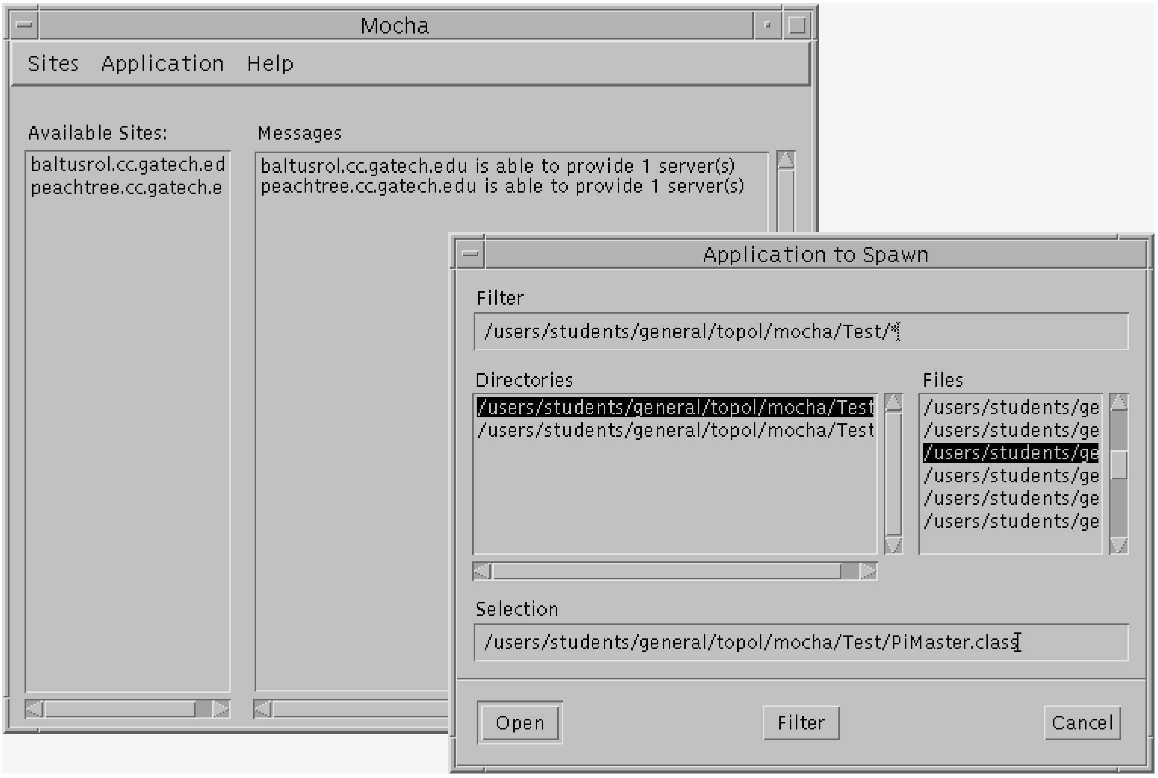

Figure 13: Spawning an application through Mocha's user interface.

as necessary. This functionality greatly reduces the difficulty associated with spawning a task at a remote site and ensuring that the appropriate object files are available at the remote location.

- *Network Communication Support*–Mocha's 100% Java custom network communication support is tailored to supporting communication activities not well suited to standard Java networking protocol support such as TCP and UDP and as a result improves the performance of fine grain message passing activities performed by the runtime.

- *Security*–The security provided by Mocha protects remote sites by recognizing when tasks are performing insecure actions and prohibiting the task from proceeding with the insecure action.

- *Debugging Facilities*–The debugging facilities are able to automatically ship debugging information back to the root site in which the application began execution. This debugging information contains a call stack trace that identifies the point of failure in the execution. This enables errors in home service applications to be troubleshooted without it being necessary for the home user to communicate to customer service centers the exact error that occurred.

- *Tools*–Mocha's tools provide support for generating host pools for a wide area computing session and illustrate that it is possible to partially automate this process.

In the next chapter, Mocha's robust statesharing facilities are presented including a description of Mocha's shared object model, basic algorithms and key implementation features. Furthermore, failure detection and handling refinements to the shared object model are also described.

Chapter 4

State Sharing

Mocha's shared object model permits threads in separate Mocha Servers to share state. The primary goals of the model are (i) to provide an efficient scheme for the consistency maintenance of shared state between threads across Mocha servers running at different nodes, and (ii) to support the sharing of complex objects.

4.1 Maintaining Shared State Consistency

Mocha's model for maintaining shared state is motivated by weakly ordered shared memory models such as Release Consistency[25], Lazy Release Consistency[38], and Entry Consistency[5]. These models have been shown to provide highly efficient state sharing implementations. In these memory models, shared memory is guarded by synchronization constructs (i.e., lock acquire and release) and is only made consistent with the most recent updates when certain synchronization points in the code are reached. Thus, in these models the shared memory may only be properly accessed between lock acquire and release synchronization points. This approach towards maintaining consistency has been shown to reach performance levels that are comparable to message passing models and thus is a strong starting point for Mocha's shared object model.

Mocha's shared object model consists of `Replica` and `ReplicaLock` objects. All objects that are desired to be shared in the Mocha system must be of type `Replica` or subclass from it. `Replicas` are not required to represent a fixed size of data; the amount of shared data contained in a `Replica` may grow and shrink as the needs of the `Replica` vary during application execution. `Replicas` may contain both homogeneous arrays of primitive data types as well as bona fide Java objects which are serializable[52]. A thread that wishes to create a shared object might use the following section of code:

```
int myarray[] = new int[10];
Replica replica1 = new Replica ("flatwareIndex", mocha, myarray, 5);
```

Here, the `Replica` constructor utilizes methods in the Mocha object `mocha` to create a shared object that is an array of integers, will have 5 replicas, and is identified by the string *flatwareIndex*. Threads that wish to acquire replicas of this object would use a similar constructor:

```
Replica replica1 = new Replica ("flatwareIndex", mocha);
```

Here it is not necessary to denote what type of shared object the replica refers to as this information is already known by the Mocha runtime. For programming purposes, the `Replica` class provides signature methods that enable the application to determine the type and amount of data the `Replica` represents.

In their generic form, `Replica` objects enable the Mocha system to replicate or *publish* shared data at either all or a subset of cooperating Mocha threads. Since multiple nodes share `Replica` objects, in order for such objects to serve in a manner that resembles a shared object, each `Replica` must be *associated* with a `ReplicaLock`

```
...

int intarray[] = new int[5];
String string = new String("Hello World");

Replica replica1 = new Replica ("flatwareIndex", mocha,  intarray, 5);
Replica replica2 = new Replica ("plateIndex", mocha, intarray, 5);
Replica replica3 = new Replica ("glasswareIndex", mocha, intarray, 5);

// The following is a generated subclass of Replica used for sharing
// a Java String,  a general purpose object.
StringReplica replica4 = new StringReplica ("text", mocha, string, 5);

// need to create a ReplicaLock, use 1 as lock id
ReplicaLock rlock1 = new ReplicaLock( 1, mocha);

// associate Replicas with lock
rlock1.associate(replica1);
rlock1.associate(replica2);
rlock1.associate(replica3);

rlock1.lock();  // Lock must be acquired and Replicas are
                // made consistent before proceeding

// Replicas may now be accessed or updated in a consistent manner
replica1.intdata[0] = 1;
replica2.intdata[0] = 1;
replica3.intdata[0] = 1;
replica4.string  = "Good Choice";
rlock1.unlock();  // Lock is now released

...
```

Figure 14: Associating and locking shared object Replicas.

object. This association is exploited to efficiently maintain consistency guarantees between `Replicas`. This approach allows Mocha's runtime to exploit the synchronization present in an application to improve the performance of consistency maintenance, a technique well established by advanced DSM systems[5, 38]. Figure 14 shows a typical section of code in which `Replicas` are associated with a `ReplicaLock` to enable them to serve as a form of consistent distributed shared memory. These type of `Replicas` could be used in the service to the home table setting coordinator application described earlier in this section. In this example, several shared index `Replicas` are provided. These shared objects would be utilized to control which images of the retail items are presented at the same time. A `String` object is also provided to allow the users of the GUI's to send comments to each other. As shown in Figure 14, these `Replicas` are associated with a `ReplicaLock`. Once the `ReplicaLock's lock()` method returns, the lock has been acquired and the `Replicas` are consistent. At this point the `Replicas` may be accessed and modified as desired. The `ReplicaLock` may then be released via the `unlock()` method.

4.2 Supporting General Purpose Java Objects

By themselves, Mocha's `Replica` objects are able to store homogeneous arrays of Java's primitive data types such as `byte`, `int`, or `double`. However, it is expected that many applications may desire to share more complex objects that are either user defined or those provided by Java itself. Examples of the latter include `java.util.Date`, `java.util.Hashtable`, *etc.* These objects are more complicated to support as shared objects because they must be serialized into a byte array to enable them to be transported over the network. Java provides an object serialization package that greatly simplifies the process of serializing these objects. Typically, only a few lines of user code are required to serialize or unserialize an object. With object serialization available, the only difficulty is to guarantee that objects are serialized before they may be sent to another thread and then unserialized when the update of the remote replica is performed.

Mocha supports the above functionality by allowing users to subclass from the standard `Replica` object and add a user specific complex object to the derived class as the object to be shared. Two methods are provided in `Replica` that may be overloaded in the derived class with serialization and unserialization code appropriate for the complex object that is desired to be shared. The Mocha runtime will automatically call these methods when it needs to marshal or unmarshal these shared objects. While creating a derived class and inserting the appropriate serialization/unserialization code into the overloaded methods is quite straightforward, it would nonetheless require extra coding effort from the application developer. To prevent this, a tool, `MochaGen`, is provided which generates a custom subclass of `Replica` which contains the object the user desires to share as well as a new custom constructor and the appropriate serialization/unserialization methods. Figure 15 presents skeleton code for `StringReplica`, a generated subclass of `Replica` used for sharing a Java `String` object. Thus, complex objects may be shared in a manner very similar to Mocha's standard `Replica` object. Figure 14 illustrates how both a generic `Replica` and derived `Replica` object might be used. Here, `StringReplica`, a new subclass of `Replica`, is used to share a `String` object that has been generated by `MochaGen`. This new subclass provides the following custom constructor:

```
StringReplica replica4 = new StringReplica ("text", mocha, string, 5);
```

Note that in the application the only difference in using a derived `Replica` object instead of the generic `Replica` is the need to call the new constructor which now uses a `String` object for the initialization of the subclassed `Replica`. Figure 14 also illustrates how both types of `Replicas` may be accessed or modified in an almost identical fashion.

It is worth noting that more experienced Java users are permitted to replace the

```
public class StringReplica extends Replica {
   ...

   String string;  // A ``complex'' object that is desired to be shared

   public StringReplica (String name, Mocha mocha, String string,
                         int numcopies) {
      // This constructor used to create a shared object

      // Marshal string object into a byte array
      // Use standard Mocha methods to distribute byte array to Replicas
   }

   public StringReplica (String name, Mocha mocha) {
      // This constructor used to get a replica of a shared object

      // call superclass' constructor, i.e., super(name, mocha);
      // unserialize received byte array and place in string object;
   }

   public  void serialize() {
       // Serialize string object (using standard Java object serialization)
       // and store in Replica's byte array
       // This method will be called automatically by Mocha runtime
       // when necessary
   }

   public void unserialize() {
       // Unserialize object (using standard Java object unserialization)
       // from Replica's byte array and load into string
       // This method will be called automatically by Mocha runtime
       // when necessary
   }

}
```

Figure 15: Skeleton code for a generated subclass of Mocha's Replica class to support a String object.

code that the `MochaGen` tool generates for serialization/unserialization with more optimized code when a priori knowledge regarding the use of objects is available. For example, assume a large object is being shared in which only a small number of integer variables might change value. Because only a small amount of state changes, much more efficient versions of the serialization routines are possible. Instead of serializing the whole object, more advanced routines might simply serialize/unserialize only the integer variables that have been modified.

4.3 Basic Object Consistency Algorithm

The implementation of Mocha's shared object replicas utilizes a daemon thread at each site manager, and a single synchronization thread at the home site. The home site in our system is simply the site at which the initial application thread executes. All of the threads mentioned above are implemented using the standard Java threads library. All objects that the application threads wish to share are registered with the local daemon thread allowing it to directly access the shared objects themselves. The daemon threads execute at maximum priority which guarantees they may interrupt lower priority application threads when necessary. This behavior permits the daemon threads to perform the transfer of shared objects to remote sites as well to accept shared object consistency updates from remote sites when necessary. The asynchrony present in this architecture allows Mocha the flexibility to dynamically configure itself during execution from a streamlined shared object system to one that can handle common failures.

The synchronization thread handles *lock* and *unlock* requests from application

Lock Method

```
if (another local thread currently has
 this lock or waiting for it)

 wait();

send synchronization thread ReplicaLockId

receive GRANT Message M from
 synchronization thread;

newVersionNumber = unpackNewVersionNumber;
versionFlag = unpackVersionFlag;

if (versionFlag == VERSIONOK)
 // We have an up-to-date version of replicas
 // and may simply return

else
 // we must wait for new version to be sent
 receive replicas from a remote daemon;
 unpackReplicas();

return;
```

Unlock Method

```
increment version numbers of replicas
send synchronization thread ReplicaLockId
 and newVersionNumber of associated objects

if (another local thread currently
 is waiting for this lock)
 notify();

return;
```

Figure 16: ReplicaLock object's lock and unlock methods which are executed by application threads.

threads. In addition, the synchronization thread directs daemon threads to perform operations such as transferring replicas to remote sites and is also responsible for deducing when such activities are necessary.

As described in the previous section, the Mocha shared object model consists of replicas which must be associated with a `ReplicaLock` object thereby supporting a variant of entry consistency[5]. When an application thread desires exclusive access to shared replicas, it calls the `ReplicaLock's lock()` method. The pseudo-code for this method is provided in Figure 16. As shown in the pseudo-code, when an application thread makes this method call, if any other *local* application threads have called this method, it must first wait until their calls have completed. After waiting, the thread sends the synchronization thread a lock `REQUEST` message which contains the identifier of the desired lock. When the lock is free, the synchronization thread responds with a `GRANT` message that contains the version number of the replicas associated with this lock and a flag which denotes whether or not a new version of the replicas need to be sent. If no new replicas need to be sent (i.e., the thread already has the most recent copy), the method may return and the application thread is free to access the replicas. If a new version of the replicas is coming, the application thread must wait for the new replicas to arrive and unmarshals them into the local copies before permitting access.

An application thread releases access to shared replicas by calling the `ReplicaLock's` `unlock()` method. This method call is responsible for sending the synchronization thread a message that indicates that the lock is being released. The method contains the identifier for the lock as well the updated version number associated with the replicas. Pseudo-code for this method is provided in Figure 16. Note that although it is possible that another local thread may be waiting for the lock, a local transfer is not permitted to insure lock acquisition proceeds in a manner that guarantees fairness.

The more complex aspects of Mocha's shared object support may be found in the operation of the daemon threads and the synchronization thread. Each daemon thread is responsible for startup activities such as initialization, and responding to requests regarding the transfer and acceptance of replicas. As shown in Figure 17, when a daemon thread receives a request for its copy of replicas, the thread identifies the replicas associated with the lock identifier it receives, marshals those replicas and sends them to the mandated destination.

```
while(true) {

   Receive message M from anyone;

   if (M.type == REGISTERREPLICA) {
     // perform startup and
     // initialization activities
   }
   else if (M.type == TRANSFERREPLICA) {
     lockId = unpackLockId();

     // unpack destination information
     host   = unpackDestinationAddress();
     port   = unpackDestinationPort();

     replicaLock = replicaLockVector.find(lockId);

     replicaLock.packReplicas();

     send packed replicas to destination;
   }
}
```

Figure 17: Pseudo-code for a daemon thread.

The synchronization thread at the home node is responsible for granting locks, queuing requests, and deducing whether a new version of replicas must be sent to an application thread. Figure 18 presents pseudo-code for the synchronization thread's operations. Upon receiving a request to acquire a lock, the synchronization thread determines if the lock exists and creates a `Lock` object if necessary. The thread then determines if the lock is free or currently owned by another thread. If the lock is not free the request is queued. If the lock is free, the acquire request is granted by sending a `GRANT` message to the requesting thread. This message contains the new version number for the replicas as well as a flag indicating whether or not new replicas will also arrive. The synchronization thread relies on the method `lastLockOwner()` to determine the value of the flag. If new replicas need to be sent, the synchronization thread sends a message to the daemon associated with the last owner of the lock and

```
while(true) {
   Receive message M from anyone;
   if (M.type == ACQUIRELOCK) {
      lockId = unpackLockId();
      if (!lockVector.exists(lockId)) {
          create new Lock object
      }

      lock1 = lockVector.find(lockId);
      if (!lock1.isFree()) {
          lock1.queueRequest();
      }
      else {
          lock1.queueRequest();
          packNewVersionNumber(lock1.returnVersionNumber());
          if (lock1.lastLockOwner() = M.source) {
              packVersionFlag(VERSIONOK);
              send GRANT message to M.source;
          }
          else {
              packVersionFlag(NEEDNEWVERSION);
              send GRANT message to M.source;
              send TRANSFERREPLICA to lock1.lastLockOwner().daemon;
          }
      }
   }
   else if (M.type == RELEASELOCK) {
      lock1 = lockVector.find(lockId);
      lock1.updateVersionNumber();
      lock1.updateDaemonId();
      //dequeue the current owner
      lockRequest = lock1.dequeueRequest();
      lock1.lastLockOwner = lockRequest.requestingHost;
      if (!lock1.isFree()) {
         nextlockRequest = lock1.examineQueueHead();
         packNewVersionNumber(lock1.returnVersionNumber());
         packVersionFlag(NEEDNEWVERSION);
         send GRANT message to nextlockRequest.source;
         send TRANSFERREPLICA to lock1.lastLockOwner().daemon;
      }
   }
}
```

Figure 18: Pseudo-code for the synchronization thread.

directs it to transfer a copy of its replicas to the application thread which desires them.

When the synchronization thread receives a request to release a lock, it updates state information such as the new version number and the identifier of the daemon that now has the most recent copy. The synchronization thread then dequeues the current owner of the lock and checks to see if another lock request is queued. If another request exists, this request is granted via a `GRANT` message and the appropriate daemon thread is instructed to transfer a copy of its replicas to the application thread which desires them.

Several aspects of the basic algorithm are worth emphasizing. First, replica data is transmitted directly from one application thread address space to another application thread without having to be transmitted via the (central) synchronization thread. This allows the system to exploit locality which may exist in a wide area distributed computing environment. Second, application threads never assume that replicas will arrive but instead examine a flag to determine this aspect. This approach provides the flexibility needed to efficiently augment the basic algorithm with advanced features such as a "push-based" replication scheme that is described in the section which follows. Third, the user may notice that version numbers are being maintained however not yet used in any significant manner. Their purpose is also described in the next section. Finally, for simplicity, we described the basic algorithm assuming exclusive locks. It can easily be modified to support shared (i.e., read-only) locks.

4.4 Fault Tolerant Refinements

As previously discussed, we anticipate failures to be more common in wide area computing environments than traditional local area network computing environments. This has motivated us to modify Mocha's basic state sharing algorithm by incorporating fault handling refinements. The failure of a remote application thread can result in the following shortcomings in the basic algorithm. First, if an application thread which has released the lock to a shared object fails before another thread has pulled a copy of the shared object, this most recent version of the shared state will be lost. Thus, the next thread that desires a copy of this shared object will not be able to acquire this most recent version of the object. Second, if a thread which currently has acquired a lock fails, then no other thread will be permitted access to the shared object and this may result in a deadlock in the system.

The object sharing system can be enhanced to provide fault tolerance using a number of different techniques. These include transactional support, server replication and virtual synchrony, and checkpoint/restart mechanisms. Our focus is on developing basic support that can be used to make the object sharing system robust with minimal overhead. As a result, we focus on the following two refinements:

- We detect failures of remote Mocha servers using timeouts and take appropriate actions to handle such failures (e.g., release lock held by a failed server).

- To ensure that a shared object's state is not lost because of a node failure, an application can choose to disseminate the object's state to multiple sites. Such dissemination is done to achieve high availability even when it is not required by the consistency protocol.

Our refinements are based on the assumption that the home node, where the synchronization thread executes, is less prone to failures because it is controlled by the user initiating the application. Thus, the focus is on dealing with failures of remote nodes.

Mocha allows objects to be replicated for state sharing. We also exploit the replicas to increase the probability that there is an operational thread that has an up-to-date copy of the desired objects. Support for updating multiple replicas of objects has been added in Mocha by permitting `ReplicaLock` objects to reconfigure themselves to employ a "push-based" update scheme. In this approach, when a thread is ready to release access to the replicas associated with a `ReplicaLock`, it may disseminate a copy of the replicas to a subset of threads that have registered to utilize these replicas. This is supported by having `ReplicaLocks` maintain state information regarding other application threads that have registered that they also desire access to the replicas. Specifically, the `ReplicaLock` keeps track of the daemon threads associated with these application threads. Recall that these daemon threads have access to the shared replica objects and may apply the disseminated updates directly. Assume that R is the number of such daemon threads that share the copy of an object.

Additionally, `ReplicaLock` objects maintain state information regarding UR, the number of up-to-date copies of the shared object. Thus, UR represents a subset of the object replicas that store the most up-to-date values of the objects. If no fault tolerance is desired, $UR = 1$ and only the node that produced the current value stores it. The new value will be sent to other nodes only when they acquire the lock associated with the object. When $UR = k$, the value will be sent to k nodes even when it is not required by the consistency protocols. The changing of R and UR (and hence the reconfiguration of the level of availability of shared objects) may be performed by either application threads or by the Mocha runtime which has access to this state information through its daemon threads.

With application threads now able to dictate the amount of update dissemination, it is necessary that the synchronization thread be cognizant of the current value of UR. This permits it to decide whether or not a new replica value must be sent to an application thread. For example, if an application thread has configured a `ReplicaLock` to disseminate its version of replicas to a subset of threads and one of these threads now desires access to these replicas, it is no longer necessary for the synchronization thread to direct a daemon thread to transmit the new version because it is already there. To permit the synchronization thread to make this type of decision, the `ReplicaLock's` `unlock()` method has been modified to include in its lock release method a set of identifiers for the daemon threads to which it has disseminated copies of the replicas. When granting the lock to next application thread that desires it, the synchronization thread consults this set to determine if the application thread requires a new copy of the replicas.

With replication support in place, it is now possible for failure resiliency to be integrated into the Mocha system. The following subsections present the modifications necessary for failure detection and handling.

Failure of Non-Lock Owning Application Thread. A failure of an application thread that does not own a lock may be detected in several ways. First, if this thread had the most up-to-date version of the replicas then the synchronization thread will contact the thread's daemon thread to perform a transfer of replicas. Assuming a fault was due either to a system crash or a local user terminating the site manager process, the daemon thread will also no longer be executing. Thus, the synchronization thread will detect the failure when the transfer message it sends to the daemon thread times out. In this particular instance, the synchronization thread handles the failure by polling other daemon threads to obtain the most recent version of the replicas available. If the new object values are disseminated to multiple nodes then in all probability (depending upon the number of failures) the most recent version of the replicas will be available and the synchronization thread may forward the replicas to the requesting thread. If the values produced by the failed site were not disseminated then the synchronization thread will only receive older versions of the replicas. It can then examine version numbers to forward the most recently available old version of the replicas. This weakened consistency may be appropriate for certain classes of applications such as service to the home whereby the home user can recognize unwanted characteristics of the old version and reapply the appropriate updates to the replicas.

A second way the failure of an application thread may be detected is when another application thread utilizes the update facilities and attempts to disseminate its version to other daemon threads. Here, we again assume that the failure of an application thread implies its associated daemon thread will also fail. When attempting to disseminate the new replicas, the send message will time out. The failure has been detected and can be handled by choosing another daemon thread at another site to receive a copy of the new version of replicas.

In Mocha, we implemented fault detection and handling for this type of failure by having the synchronization thread detect failures when the transfer message it sends to a daemon thread times out. The synchronization thread then handles the failure by polling other daemon threads to obtain the most recent version of the replicas available.

Failure of Lock Owning Application Thread. Failure of an application thread that currently owns a lock is another situation that must detected and handled. This failure may be detected by having the synchronization thread timestamp lock acquisitions and by requiring that threads indicate approximately how long they need to hold a lock. The synchronization thread can periodically peruse its list of held locks to determine if any threads are holding locks for an extraordinary amount of time and therefore a candidate for being a failed thread. The synchronization thread can confirm this suspicion by sending a "heartbeat" message to the appropriate daemon thread. If this message times out, the synchronization thread can assume the application thread has failed and thus the failure has been detected. Here, the synchronization thread can simply break the lock and give it to the next application thread that desires it. This occurs even if the holder of the lock is still alive. The approach is very similar to the concept of *Leases* described in [14]. After breaking the lock, the most recent copy of the replicas will now be those available from daemon thread of the previous owner of the lock or if necessary, the synchronization thread may resort to polling daemons for the most recent version of the replicas. Furthermore, an application thread that fails in this manner is prevented from making future requests.

In Mocha, we implemented fault detection and handling for this type of failure by having the user threads indicate how long they need to hold a lock. The synchronization thread timestamps lock acquisitions and if a lock is held for longer than expected, the synchronization thread breaks the lock and gives it to the next application thread that desires it. The next application thread to acquire the lock receives the most recent version of the replica available.

Failure of Synchronization Thread. In general, we assume the synchronization thread executes at the initial home site and therefore will be less likely to fail compared to application threads executing at remote sites. Failures that take place when a machine reboots, a task is killed by another user, or failures due to network contention are not expected to happen very frequently at the home site. This is due to the fact that in the local environment there is more control over the computing resources. Because of these reasons and to keep the overhead of failure handling low, we chose not to implement the synchronization thread in a failure resilient manner. However, we do have some ideas on how to mitigate failures of the synchronization thread. Failure detection and handling of the synchronization thread could be handled by logging its state and employing a recovery protocol whereby a new synchronization thread is spawned which informs the daemon threads of its existence. Application threads which time out attempting to contact the failed synchronization thread can query the local daemon thread to obtain the location of the newly created surrogate synchronization thread.

Mitigating Failures Occurring During State Transfer. Failures that occur during the process of transferring shared state between tasks must also be mitigated.

For example, when a task acquires a lock, the synchronization thread directs the task that previously held the lock to transfer the shared state to the task that has acquired the lock. Should the task that previously held the lock fail after being informed to perform a shared state transfer, but *before* the task has completed the transfer, the task waiting to receive the new replicas will wait indefinitely. To prevent such blocking from occurring during the state transfer, timeouts are used to detect when a shared state transfer has not occurred. When such a timeout occurs, the task sends a `NEEDNEWREPLICA` message to the synchronization thread. The synchronization thread then utilizes a polling mechanism to locate the most up to date copy available. It then directs the daemon thread at the site at which the most up to date copy of the replica exists to perform a state transfer to the task that is waiting for the replicas. Currently, the system uses a 15 second timeout value for detecting when state transfer has not occurred. We have found this value to be appropriate for the environments we have experimented with. This however may need to be reconfigured for other environments.

Disseminating Failure Information. Once a task has detected the failure of another task, it is important to disseminate this information to tasks that are still functioning. This enables these tasks to become aware of the failure without themselves using timeouts to detect the failures and thus avoid unnecessary overhead. To support this functionality, the `lock()` and `unlock()` methods associated with the `ReplicaLock` class as well as its counterparts in the synchronization thread have been modified to include a set of identifiers for daemon threads detected as having failed. This permits state transfers and polling mechanisms to avoid contacting daemon threads already known to have failed.

Supporting a Quality Contract for Replica Transfers. For some applications, it may be perfectly suitable to receive an old version of replicas from a transfer. This situation can occur if enough failures have occurred that the most recent version of the replicas is no longer available. For other applications, it may be imperative that the task must receive the most up to date version of replicas to continue executing. To allow applications to develop their own policy regarding what type of replica transfer is permissible, the `ReplicaLock` class `lock()` method returns a value that informs the application thread whether or not it was possible to transfer the most recent version of the replicas. The application may then use this information to decide what course of action it should take at this point.

4.5 Evaluation

This section provides an evaluation of the state sharing support provided by our prototype system. We present the overheads associated with several core activities performed by the system to support object sharing. These include lock acquisition latency, data marshaling overheads, and communication overheads incurred transferring replicas in local and wide area networks as well as in the Georgia Tech Home Information Infrastructure Lab, a prototype home service environment that includes a Windows 95 PC connected to a Unix workstation hub via a digital subscriber line (ADSL) modem.

Table 3 presents the amount of time in milliseconds to perform lock acquisition

Local Area Network (Fast Ethernet)	5
Wide Area (Internet)	19
Windows 95 PC Connected by ADSL Modem	16

Table 3: Time to acquire a lock (with no data transfer) in milliseconds.

using the methods of the `ReplicaLock` object. The local area network results are from two SUN ULTRA 1 machines on Fast Ethernet. The wide area results are from a SUN ULTRA 1 and a SUN SPARCstation 20 connected via the Internet separated by a distance of approximately 6 miles. The home service environment results are from a Windows 95 PC to a SUN ULTRA 1 machine. These machines are physically located across campus from each other and are connected by way of a Unix workstation which serves as a hub. The Windows 95 PC is connected to the hub with an ADSL modem and the hub is connected to the SUN ULTRA 1 via an ATM network. As shown in the Table, lock acquisition latency in wide area networks can be significantly greater than the latency experienced in local area networks.

Figure 19 presents the time for a SUN ULTRA 1 to marshal a replica into a byte array to enable it to be transferred over the network. As shown in the Figure, this activity can be somewhat expensive for large replicas. This inefficiency is a result of the Mocha system relying on the generic data marshaling constructs provided by Java JDK 1.1. These constructs utilize dynamic arrays and marshal a single byte at a time. These factors result in marshaling being a relatively costly operation. In the future, we plan on providing a custom marshaling library that is more efficient for our needs.

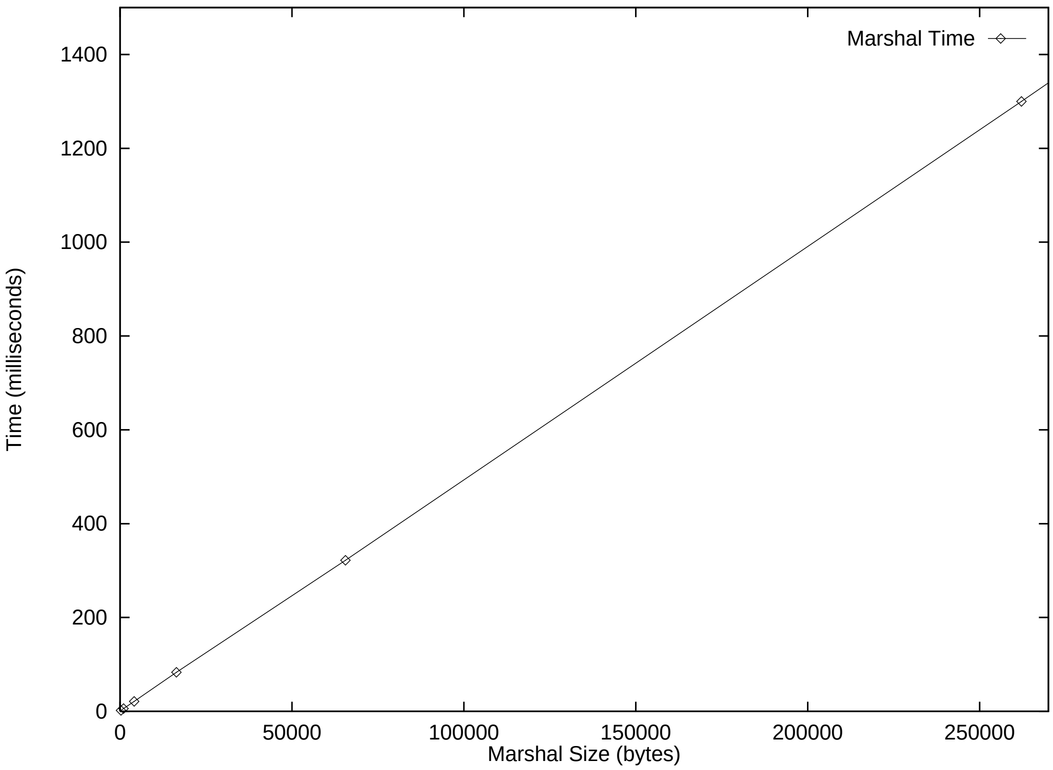

Figure 19: Time to marshal Replicas in milliseconds.

For the transfer of replicas between hosts, we have developed two separate prototype systems. In the first system, all communication is performed using Mocha's network object library. This library implements reliable and sequenced delivery of messages and performs fragmentation and reassembly. It is scalable in the number of hosts that communicate with the library because it performs its own upward multiplexing of packets. It is particularly well suited for sending small messages as it avoids the heavy connection and tear-down overheads associated with other transport protocols such as TCP. Empirically, we have found Mocha's network communication library to be approximately twice as fast as TCP for sending small (i.e., less than 256 byte) messages.

For the second prototype, small "control" messages used for lock acquisition and directing data transfers are sent using Mocha's network object library. For the actual transfer of replica data which typically involves sending large messages, we have developed a "hybrid protocol" approach whereby Mocha's custom network object library is used in conjunction with TCP. Here, Mocha's network communication is used for establishing a TCP connection (i.e., propagating TCP port numbers) and the actual transfer of replica data is done using TCP. Figure 20 presents the times in milliseconds to disseminate a 1K replica to several local area network hosts and Figure 21 illustrates the same dissemination of replicas for wide area networks. In both environments, solely using Mocha's network communication library is the more efficient approach. This is attributable to the higher connection and tear-down overheads associated with the hybrid approach.

As we increase the size of the replicas, the hybrid protocol approach provided by the second prototype begins to perform much better than simply using Mocha's

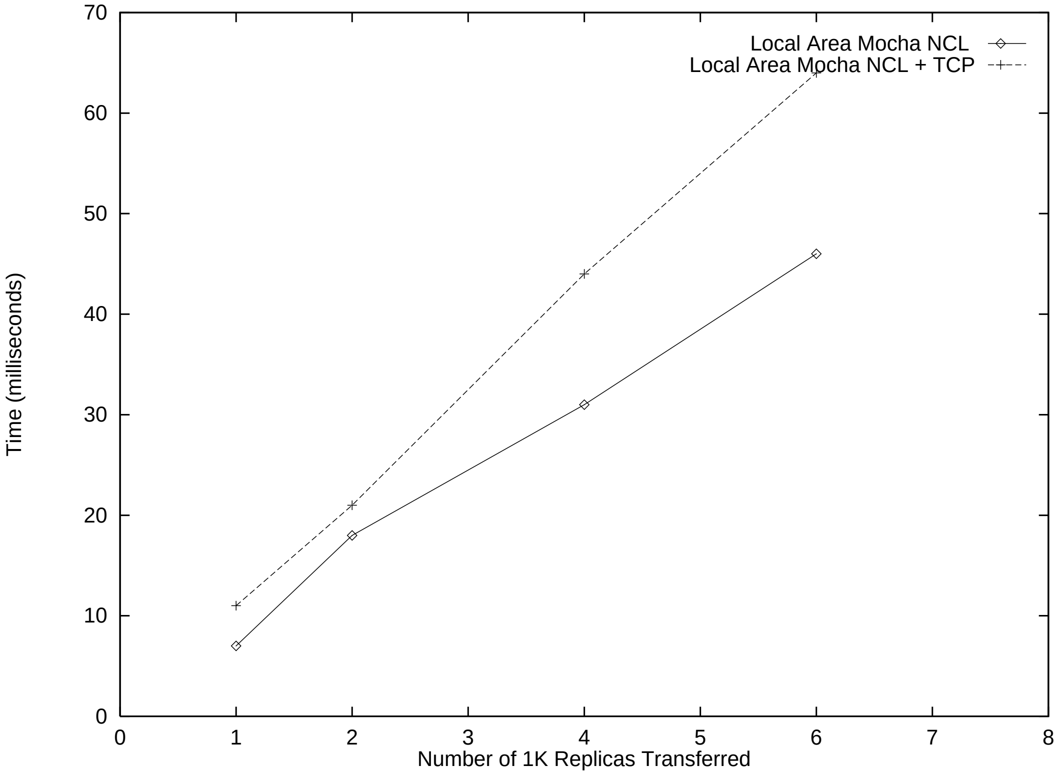

Figure 20: Time for local area transfer of 1K replicas in milliseconds.

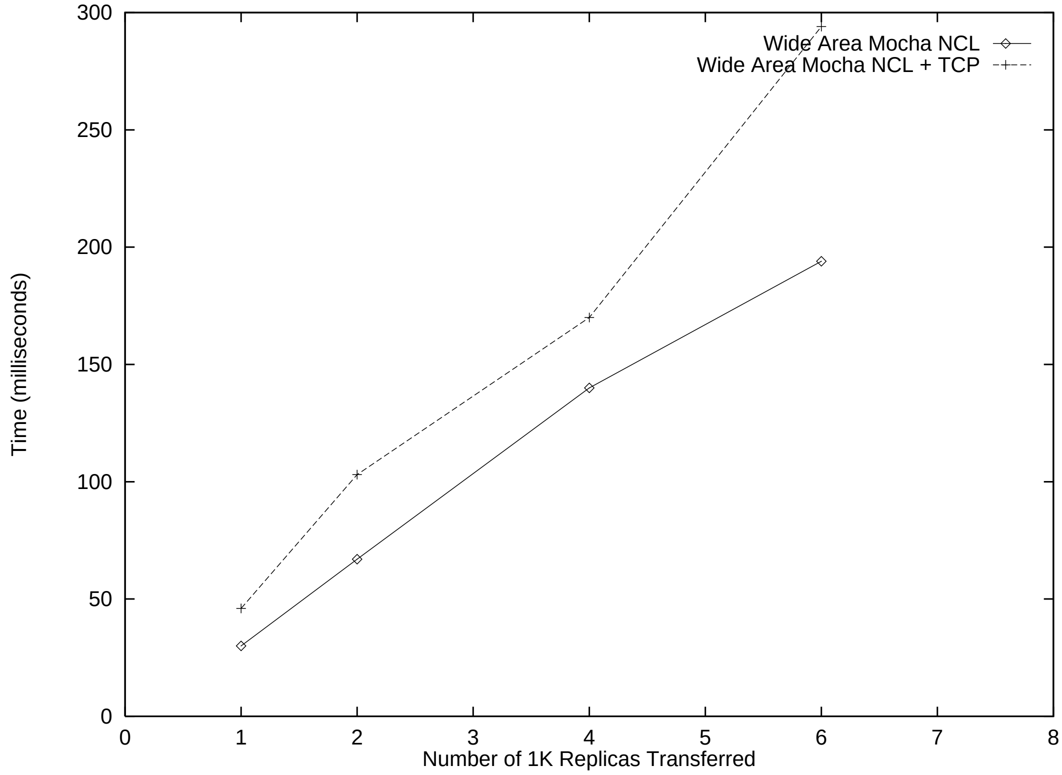

Figure 21: Time for wide area transfer of 1K replicas in milliseconds.

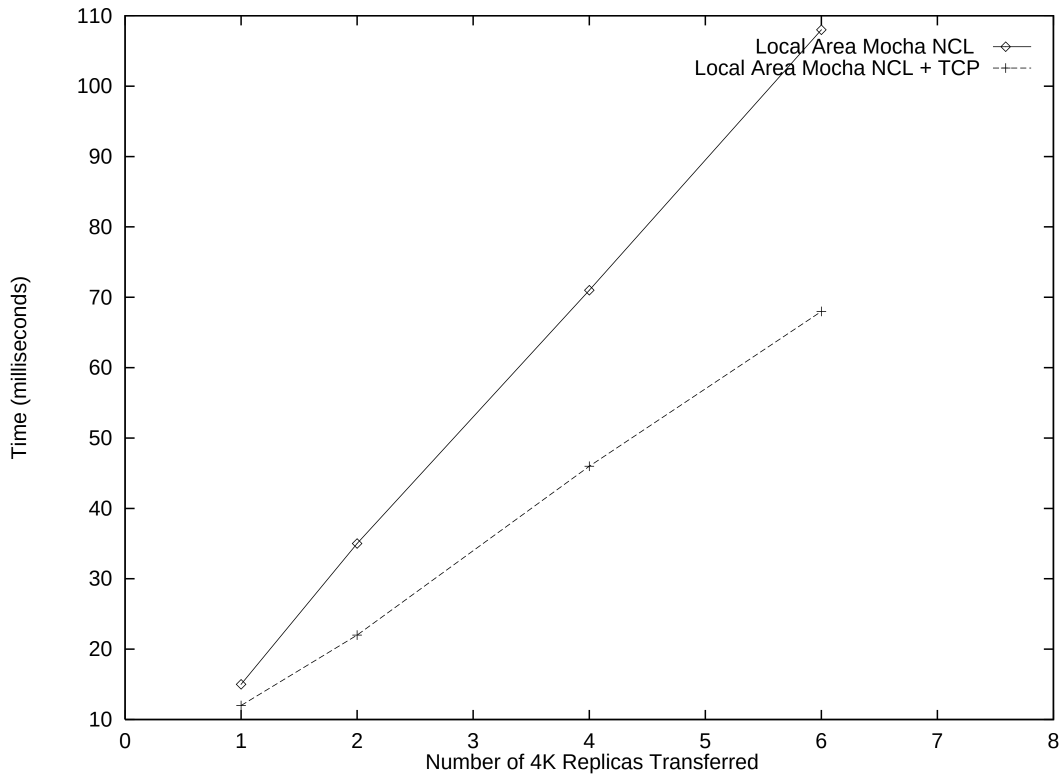

Figure 22: Time for local area transfer of 4K replicas in milliseconds.

network communication library for all network environments investigated. Figure 22 presents the results for 4K replica transfers in local area networks, and Figures 23 and 24 illustrate the results for wide area and home service networks, respectively.

Furthermore, as shown in Figures 25, 26, and 27, for larger replicas reaching sizes of 256K, the superiority of the hybrid protocol becomes clear. This vast improvement is attributable to the speeds at which both protocols perform fragmentation and reassembly. Mocha's fragmentation and assembly executes at user level running as interpreted byte code. The TCP fragmentation and reassembly executes as native binary code at the kernel level. This vast disparity of execution speeds allows TCP to easily ameliorate its connection and tear-down overheads for large, multipacket

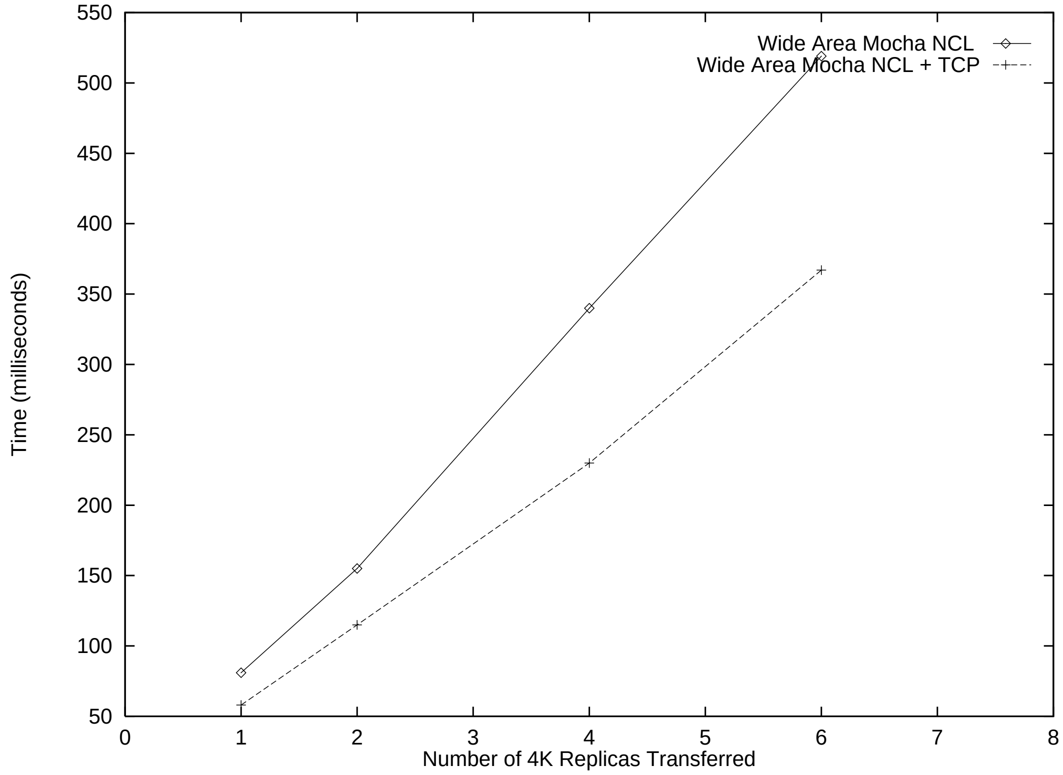

Figure 23: Time for wide area transfer of 4K replicas in milliseconds.

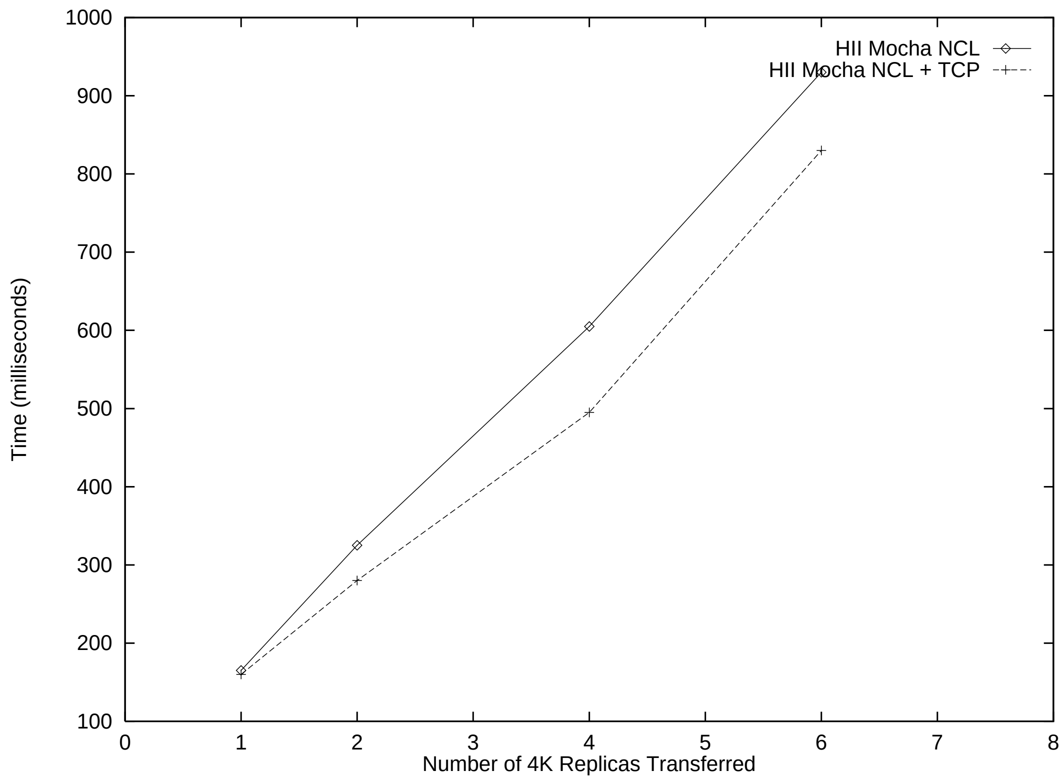

Figure 24: Time for home service transfer of 4K replicas in milliseconds.

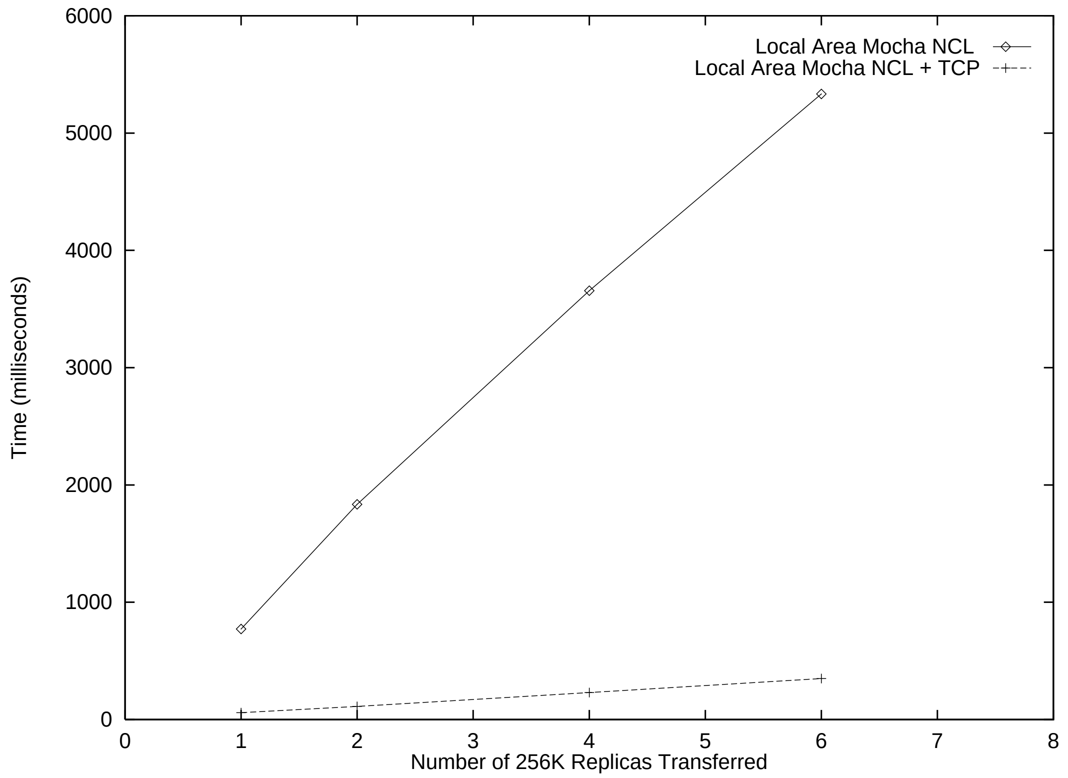

Figure 25: Time for local area transfer of 256K replicas in milliseconds.

messages.

Another interesting question that deserves investigation concerns the point at which the hybrid protocol becomes more efficient than the standard protocol and whether or not this point is different for each of the three environments. As illustrated in Figures 28, 29, and 30, the point at which the hybrid protocol becomes more efficient than the standard protocol when disseminating replicas to three sites is between 1500 and 2500 bytes for all three environments. Furthermore, this point remains more or less constant as the number of disseminated replicas is increased from three to six.

Another mechanism utilized in the Mocha system is the ability to poll a host for

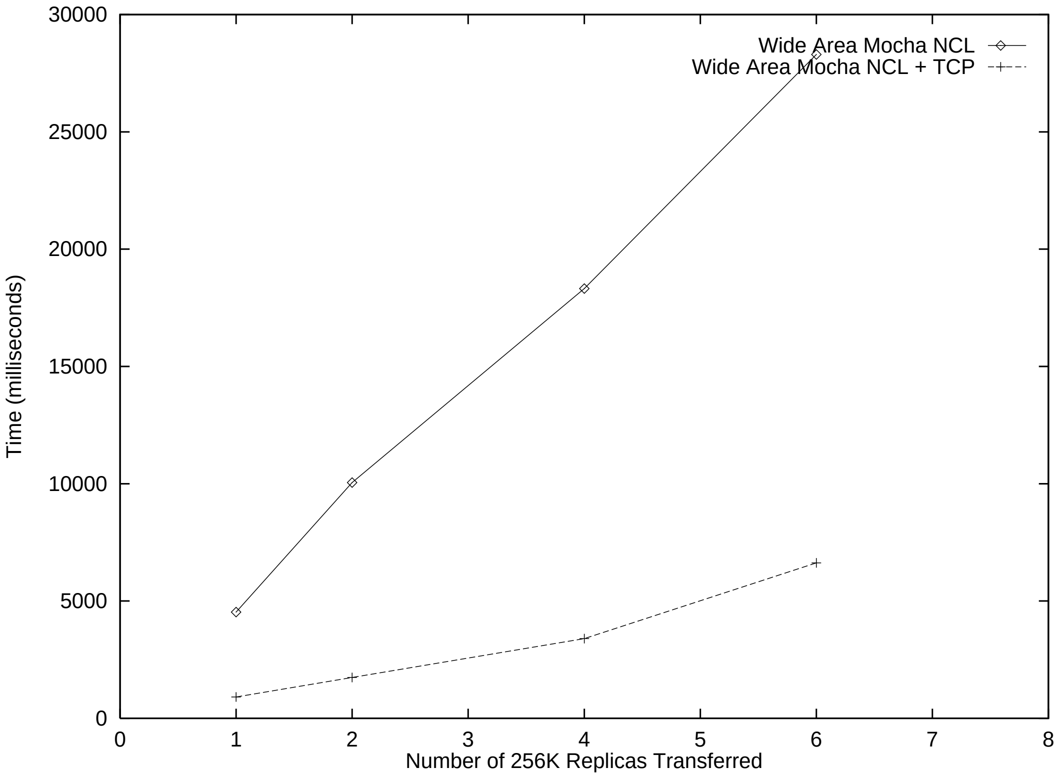

Figure 26: Time for wide area transfer of 256K replicas in milliseconds.

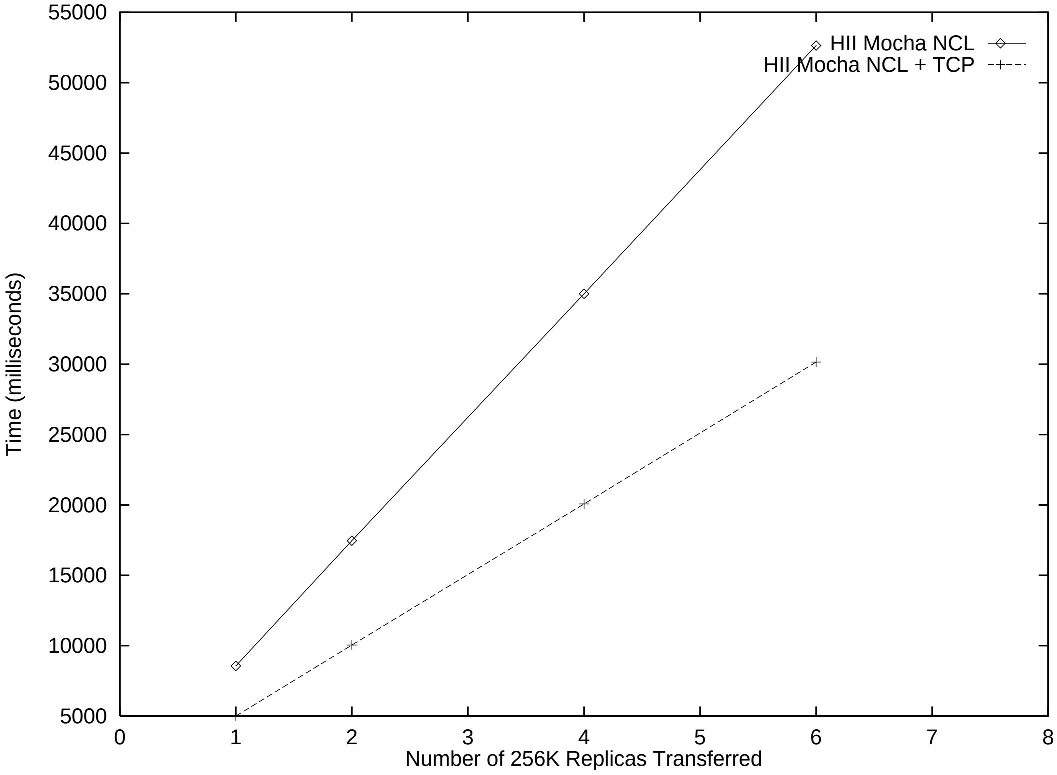

Figure 27: Time for home service transfer of 256K replicas in milliseconds.

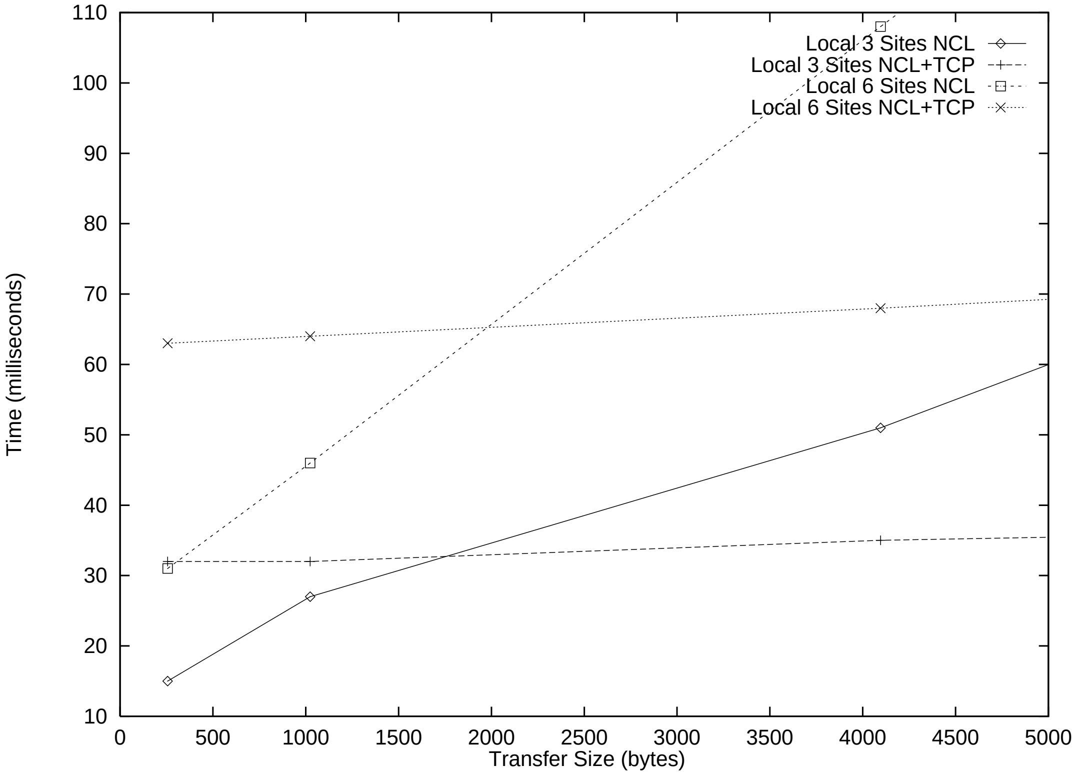

Figure 28: Times for local area transfer of replicas for Mocha NCL and hybrid communication substrates.

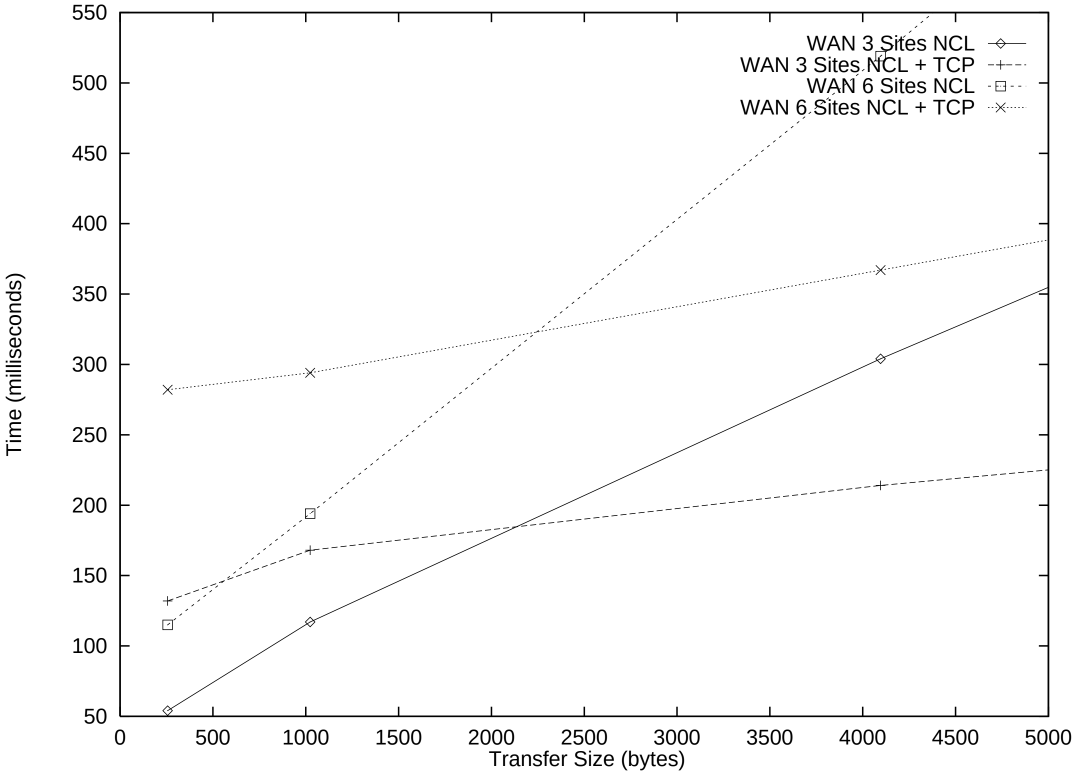

Figure 29: Times for local area transfer of replicas for Mocha NCL and hybrid communication substrates.

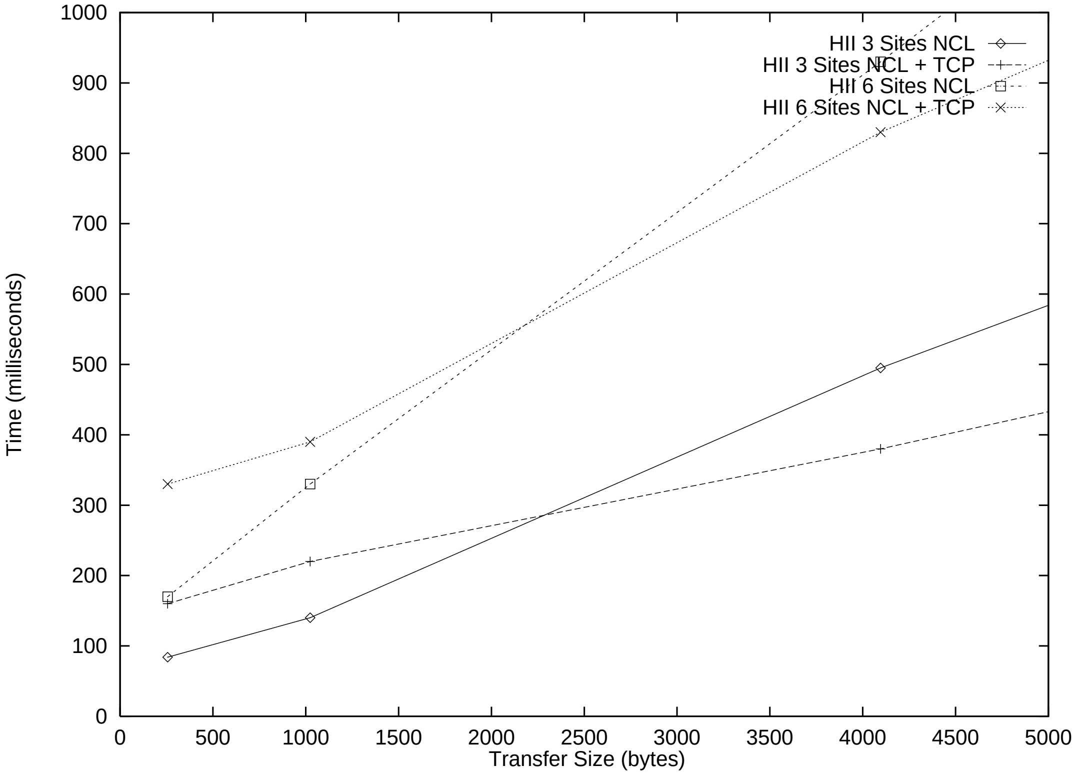

Figure 30: Time for home service transfer of replicas for Mocha NCL and hybrid communication substrates.

Local Area Network (Fast Ethernet)	8.5
Wide Area (Internet)	25
Windows 95 PC Connected by ADSL Modem	19

Table 4: Time to poll an individual host in various environments in milliseconds.

the version number of its replica. This is utilized during failure recovery to determine the most up to date replica available. Table 4 presents the overhead in milliseconds for polling an individual host in various networking environments. As illustrated in the Table, polling is a relatively low cost operation, especially considering that it is only used during failure recovery periods.

In summary, several aspects regarding the cost of update dissemination for high availability as well as the relative efficiency of the two approaches for transferring replicas are apparent. As can be seen from Figure 23, when the number of moderately sized (i.e., 4K) replicas that are maintained up-to-date in a wide area environment is increased from 1 to 2, the overhead for consistency maintenance approximately doubles. Furthermore, as also depicted in Figure 23, the hybrid protocol approach can result in an improvement of approximately 30% over Mocha's basic protocol for transferring replicas as small as 4K to multiple (i.e., 6) sites. As shown in Figure 26, for replicas as large as 256K, the hybrid protocol can reduce transfer costs by as much as 70% over the basic protocol when transferring replicas to multiple sites in a wide area environment. Finally, as shown in Table 4, Mocha's mechanism for pulling for replica version numbers is a relatively low cost operation.

4.6 Related Work

Mocha's goals as well as the techniques employed by it are related to several research areas. These include metacomputing systems, DSM systems, and systems that address fault-tolerance. We describe systems from each of these categories.

Wide area computing is closely related to metacomputing. A number of metacomputing systems are currently being implemented. Some systems rely on message passing instead of shared objects to allow tasks to cooperate. Examples of these systems include Chandy's worldwide distributed system[12] and IceT[26]. For the Mocha system, we chose to utilize shared objects instead of message passing because they provide a model that is simpler to program than standard message passing. Furthermore, we provide replicated copies for failure handling which in a message passing model would require group communication and virtual synchrony support[7] and is not currently addressed by the above message passing systems.

Another non-shared object approach utilized by metacomputing systems to allow task cooperation is the use of remote procedure call (RPC) or its object-based equivalent, remote method invocation (RMI). Atlas[3], WebWork[21], NetSolve[11], and Legion[27] all rely on forms of RPC/RMI for task interaction. In some cases, a RPC/RMI model's performance suffers from the clients need to repeatedly contact a server to perform distributed computation. This of course depends on the type of remote computing activities being performed as well as the type of caching strategies employed by the RPC/RMI system. For a more thorough comparison of RPC/RMI and shared memory please refer to [67].

Several metacomputing systems are currently providing shared memory. The TIE design[16] supports shared objects via object caching and entry consistency. The developers of TIE believe that in the future, available network bandwidth will be limited (due to growing popularity of the Internet) and aggressive caching must be performed to avoid server bottlenecks. The TIE design is in some ways similar to Mocha but the emphasis of the two systems is quite different. the TIE system currently focuses on mobile objects and security while Mocha focuses on high availability and advanced support for sharing general purpose Java objects. We are also not aware of an implementation of TIE.

The ParaWeb system[10] modifies the Java interpreter to provide a global shared address space using distributed shared memory techniques pioneered by systems such as Munin and Treadmarks. In the ParaWeb implementation, the Java interpreters have been modified to permit them to cooperate and maintain the illusion of global shared memory. ParaWeb utilizes Java's built-in synchronization facilities to monitor when remote memory must be updated to maintain the illusion of consistent global memory. In a similar approach, Yu and Cox[67] are currently implementing a parallel Java Virtual Machine layered on top of the TreadMarks page-based distributed shared memory system. Currently, they are addressing problems such as data type conversion between machines of different architectures as well as garbage collection. Mocha's approach towards supporting shared state differs from these two systems as it supports it at the object level while these systems support sharing at the page level. The difference in the two approaches results in the need to solve different types of problems. With the shared sequence of bytes approach, these systems typically must modify the Java virtual machine, must compensate for different byte orderings in heterogeneous environments, and mitigate false sharing. Although Mocha's shared object approach does not encounter these types of problems, it must deal with issues such as how to support complex objects.

Java Shared Data API (JSDA)[36] provides shared variable support using its own multipoint data delivery service. With JSDA, an update to a shared variable is sent to a session server which then sends the update to other threads that are sharing the variable. In contrast, the Mocha system attempts to exploit locality by sending shared state changes directly to the next thread that needs access to the data. Mocha also allows the number of updated replicas to be configured whereas JSDA updates all copies.

PageSpace[15] relies on a Linda-like coordination technology. Essentially, PageSpace supports a global tuple space which nodes may insert or remove tuples without any regard for where the tuples are stored.

Mocha's consistency actions are driven by synchronization operations. In certain systems and applications, synchronization for all operations may not be desirable. For example, systems such as Bayou[61], Coda[39], and Rover[35] which address mobility, avoid synchronization and instead rely upon conflict detection and resolution to maintain consistency. Our future work will explore non-synchronization based consistency models that are suitable for supporting shared objects.

In summary, Mocha's state sharing support distinguishes itself from the above systems by combining advanced distributed shared object techniques with failure handling support that allows its overheads to be controlled based upon the level of availability needed for shared objects. Additionally, Mocha's runtime exploits Java's method overriding and serialization capabilities to support the sharing of complex objects without requiring modifications to the standard Java interpreter. Moreover, as discussed in Chapter 5, we have a broad application focus that includes applications directed to the home.

4.7 Discussion

In this chapter, we have presented a robust shared object model for wide area distributed applications that has been implemented as part of the Mocha wide area computing infrastructure we are currently developing. Our model provides support for shared objects on heterogeneous platforms, and utilizes advanced distributed memory techniques for maintaining consistency of shared objects. Moreover, our system provides fault tolerance support that allows its overhead to be controlled based on the level of availability needed by an application. We have investigated a combination of communication protocols approach for improving the efficiency of shared state transfer between hosts and performed an empirical evaluation of the basic costs of two versions of our prototype system in local area, wide area, and home service networks.

Chapter 5

Applications

In this chapter we explore several classes of applications that Mocha is able to support. The first class is electronic commerce home service applications. These applications execute in heterogeneous environments that include archetypical home computing platforms such as a PC running Windows 95 operating system. For this class of applications, we show that Mocha is able to support an interactive electronic commerce application utilizing its state sharing facilities to enable collaborative home shopping.

The next class of applications we investigate is scientific applications suitable for execution in a wide area computing environment. We illustrate how the traveling salesperson problem, a parallel branch and bound application that requires state sharing, may be developed in a failure resilient manner using the mechanisms provided by Mocha. We also provide empirical results for this application that illustrate the benefits of the dissemination capabilities provided by Mocha as well as the overheads incurred by its failure detection and recovery mechanisms.

A third class of applications we explore is traditional collaborative applications such as a shared calendar. These applications in some circumstances are able to exploit a weaker notion of shared state consistency when available. We illustrate how Mocha's mechanisms may be utilized to provide a variant of weakened consistency and how using Mocha in this manner can yield significant performance improvements

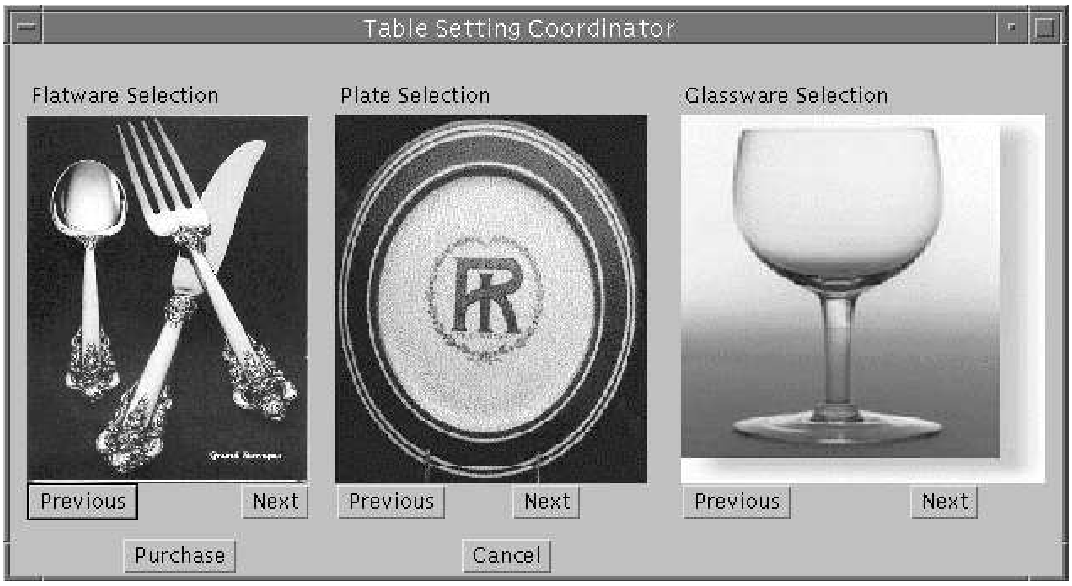

Figure 31: Mocha table setting coordinator home service application.

over collaborative applications that use Mocha's mechanisms in a more tightly coupled fashion.

5.1 Electronic Commerce Applications

Figure 31 presents a sample interactive home service application written with the Mocha system that utilizes the system's state sharing mechanisms to support collaborative home shopping. The application is a formal dinner table setting coordinator application similar to the one described in Chapter 3. As discussed in Chapter 3, in this home shopping scenario, a consumer residing at home wishes to add a new formal dinner table place setting composed of flatware, plates, and glassware. At the consumer's home, a graphical user interface (GUI) is executing which allows various flatware, plates, and glassware to be viewed together so that the consumer may "mix and match" these items and end up with a pleasing coordinated table setting. Additionally, a sales associate located at the retail outlet may also have a copy of the graphical user interface which permits the associate to see what the customer is selecting and may suggest alternatives that are then presented in the customer's GUI. Furthermore, the home consumer may have requested friends located at other homes to also participate in this decision making and therefore they too may be running a GUI and viewing the possibilities and also making suggestions. In this scenario, it is expected that the platforms on which the GUI executes in each of the homes would be vastly different from the platform at the retail outlet.

In this application, the GUI shown in Figure 31 is sent via Mocha's remote evaluation support to execute at several remote sites. Each site may modify the flatware, plates, or glassware currently being displayed by pushing the appropriate `previous` or `next` button. These buttons result in activating callbacks which modify shared index variables associated with each item. A thread which executes in each remote GUI periodically polls the shared index variables for new values and updates the local display as needed. The graphical images are also shared as replicas but are not associated with a `ReplicaLock`. Thus, they are cached at each host without any consistency maintenance being performed on them. The shared indexes do however rely on the system's consistency maintenance facilities.

Figure 14 in Chapter 4 illustrates how the shared index variables for this application may be created, associated with a `ReplicaLock`, and modified. Figure 32 illustrates the sharing and platform independent displaying of flatware images which are also critical aspects of this application. As illustrated in the Figure, we first create `flatwareImage[]` which is an array of the abstract class `Image` Java provides to hold

```
...

Image flatwareImage[] = new Image[3];

Replica flatwareReplica[] = new Replica[3];

for (i = 0; i < 3; i++) {
  flatwareReplica[i] = new Replica ("flatware"+i, mocha);
}

for (i = 0; i < 3; i++) {
  flatwareImage[i] = Toolkit.getDefaultToolkit().createImage(flatwareReplica[i].bytedata);
}


ImageDisplayer flatwareDisplayer  = new ImageDisplayer(flatwareImage);

...
```

Figure 32: Creating shared image replicas.

images in a platform independent fashion. We then create an array of `Replica` objects used to reference the shared images. Mocha's standard mechanism for acquiring an initial copy of a shared object is then used:

```
for (i = 0; i < 3; i++) {
   flatwareReplica[i] = new Replica ("flatware"+i, mocha);
}
```

In Java, images are stored as `byte` arrays. Mocha supports the sharing of `byte` arrays as one of its default data types. Thus, it is not necessary for this application to create custom shared object replicas using Mocha's `MochaGen` utility. Instead, we simply share the images as `byte` arrays. In order to convert these arrays into an image which may then be referenced using Java's `Image` class, Java provides a method, `createImage(byte[] array)`, as part of the default `Toolkit` class that permits `byte` arrays to be converted into bona fide Java `Image` objects. This is performed by the application in the following fashion:

```
for (i = 0; i < 3; i++) {
   flatwareImage[i] = Toolkit.getDefaultToolkit().
        createImage(flatwareReplica[i].bytedata);
}
```

At this point, the `Image` objects are passed as a parameter to the constructor of the class `ImageDisplayer` which subclasses Java's `Canvas` class to enable images to be displayed on the screen.

5.1.1 Performance

With respect to performance of this application, we have measured the marshaling, lock acquisition, and transfer costs of keeping these three replicas consistent in a prototype home service environment. The home service environment results are from a Windows 95 PC connected to a SUN ULTRA 1 machine. These machines are physically located across campus from each other and are connected by way of a Unix workstation which serves as a hub. The Windows 95 PC is connected to the hub with an ADSL modem and the hub is connected to the SUN ULTRA 1 via an ATM network. The application performs shared state transfers of three `Replicas`, each of which stores an integer index variable. In this environment the transfer costs for sharing these replicas were as follows: marshaling required 2 milliseconds, lock acquisition and release overhead was 31 milliseconds and transfer costs were measured at 38 milliseconds. Overall, the total cost of maintaining consistency is 71 milliseconds, a latency value that we feel is suitable for this type of application.

5.2 Scientific Applications

Scientific parallel computing applications are another domain in which the functionality provided by Mocha may be beneficial. Obviously, when possible, it is best to execute a parallel computing application in a low latency network environment using a non-interpretive language.

In some cases, however, it may be necessary to execute a parallel computing application in a wide area environment. For example, an application may require scientific data that is only available at a remote site. Furthermore, sufficient computing resources may not be available locally and therefore remote sites must be utilized to obtain the desired amount of computing cycles. Finally, as techniques such as just in time compilation[17, 55] that improve the performance of interpretive languages become more mature, languages such as Java may become an acceptable platform for some types of parallel applications.

When developing parallel computing applications that are intended to be executed in wide area computing environments, a significant starting point is to begin with implementation strategies developed for cluster environments. These environments are in many ways similar to wide area computing environments. For example, both environments are composed of workstations that have varying configurations in terms of CPU speed and memory. Furthermore, these workstations are part of an open network computing environment and both the workstations as well as the network may be subject to uncontrollable external loads. These factors often result in load imbalances and dynamic fluctuations of delivered resources which can be a major source of performance degradation for a parallel application[53].

One strategy for developing network based concurrent computing applications shown to be particularly effective is use of an *agenda parallel* model that utilizes dynamic load balancing to combat the effect of load imbalance[53]. In this model, the application is implemented using a "bag of tasks" algorithm where pending work is in the form of a queue of tasks maintained by a master process. The master process is responsible for assigning tasks to processing elements as they become idle and also for collecting results. With this scheme, more work is assigned to processing elements with lighter loads and this dynamic load balancing helps to mitigate the performance degradation that occurs when executing an application in a network based concurrent computing environment.

In this section, we investigate the development of a bag of tasks implementation of the traveling salesperson problem (TSP) suitable for execution in a network computing environment using the functionality provided by Mocha. This parallel application allows us to investigate the performance of Mocha's state sharing ability to serve as a distributed shared memory (DSM) facility for parallel applications. It is readily apparent that overall execution time will not be competitive with traditional implementations of TSP as Mocha applications execute at interpreted byte code speed. However, it is interesting to see if our implementation has the same relative performance characteristics as a typical DSM based parallel implementation of TSP.

We also use this application to show how it is possible to utilize Mocha's functionality to develop a failure resilient version of TSP using locks that have timeout values to permit failure detection. Furthermore, we utilize timeout values that are large enough to be suitable for wide area execution. All experiments that follow, however, are executed in a local area network as it provides a more controlled environment.

This permits the behavior of the TSP application to be characterized as precisely as possible. The next section provides an overview of the implementation of TSP for network computing using Mocha as well as the extensions that are necessary to develop a failure resilient version of the application. We then evaluate the performance of both versions of the application.

5.2.1 Implementing TSP

The TSP algorithm implemented by us is similar to the one used by Bal *et al*[2] as well as Kohli *et al*[40]. The application creates a statically defined job queue that is shared between tasks. The initial queue is composed of partial tours that are generated by unraveling possible paths through the cities to a fixed depth. Locks are utilized to dequeue jobs from the queue as well as to update the value of the best tour. Thus, there are two structures which must be shared between tasks; the job queue and the best tour value. The best tour value is checked frequently to insure that the tour currently being investigated by a task has the potential to become the new best tour. Tasks constantly read the best tour value and these accesses are permitted without acquiring a lock and hence data races can occur. In contrast, however, all updates to the best tour value are protected by a lock. The ability to support push-based dissemination when updating the best tour value has the potential to improve the performance of the application. This is because receiving an updated tour value may result in a task realizing that the tour it is investigating has no chance of being the best tour and completing the investigation would be wasteful. Instead, the task can choose the next available partial tour job and investigate it.

Developing a Failure Resilient TSP. A failure resilient TSP application can be created from the algorithm described above by combining Mocha's mechanisms for failure recovery (i.e., replication, breakable locks) with some simple algorithmic modifications that permit the application to (i) become aware of failures, and (ii) take appropriate action to reproduce the work that was lost during the failure.

We utilize Mocha's mechanisms for failure recovery by using the `setLockHoldTime()` method of class `ReplicaLock` in order to set a timeout value for the lock used by the application. That is, we enable the synchronization server to break a lock if it has been held for a duration of time that is much longer than the period of time it should take for a non-failed task to release the lock. Additionally, we modify the application to use Mocha's update dissemination capabilities to insure that should a failure occur, it will be possible to find and up-to-date copy of the shared job queue as well as the best tour value.

In order to be able to detect a failure as well as to determine which jobs were not completed because of the failure, we must disseminate to all tasks which job each task is currently investigating. Thus, we add a new shared object whose purpose is to store the identifier of the current job a task is working on. We refer to this as a *will* structure as it essentially denotes a request that the task desires some other task to complete should this task die (i.e., fail). When a task chooses a new job it marks it in its will structure and when it completes the job, the will structure is cleared. This clearing of the will structure is performed *after* the best tour value has been updated (if necessary). At the end of execution, the will structures for all tasks are examined to determine if there exists any jobs that have not been completed. These jobs are then performed by the non-failed tasks a second time to their full completion. In the

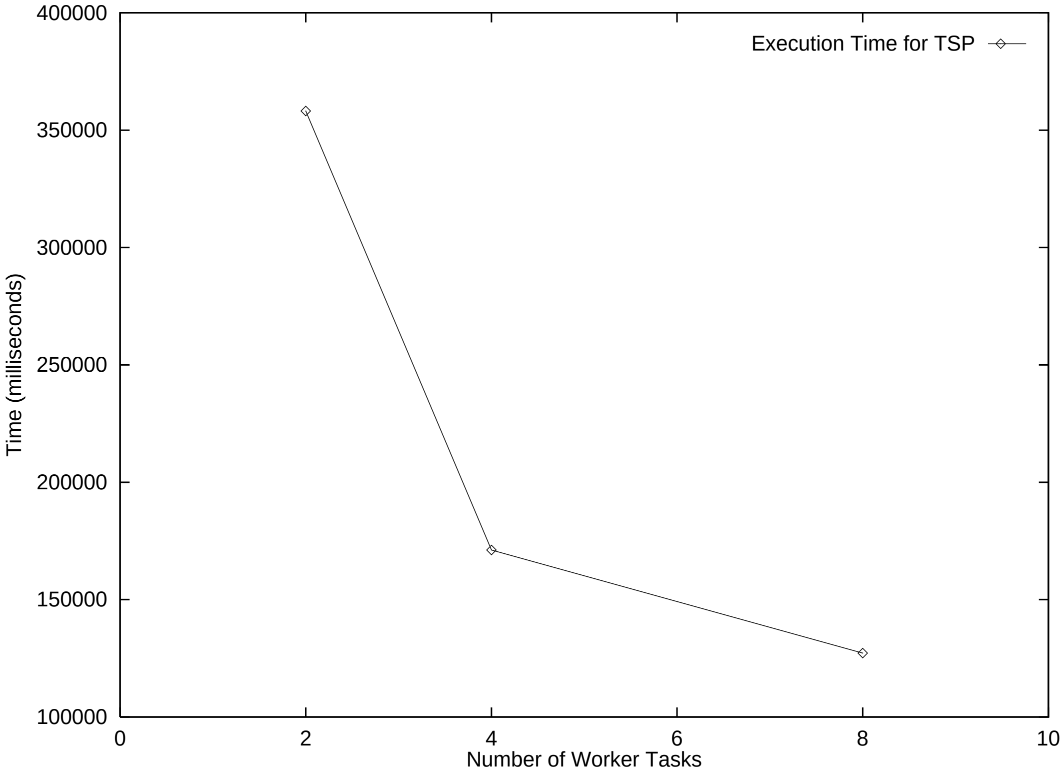

Figure 33: Completion times of TSP (no dissemination) in milliseconds.

next section, we examine the performance for both the basic implementation as well as the failure resilient implementation of TSP.

5.2.2 Performance of TSP

Figure 33 presents the completion times in milliseconds for TSP with a problem size of 17 cities when it is executed on a local area network cluster of SUN ULTRA 1 workstations. As illustrated in the Figure, completion times decrease as the number of worker tasks is increased. Thus, significant speedup is achievable with the system. Furthermore, the graph of completion times for this problem size of TSP is remarkably similar to graphs of completion times of other systems[40] that implement a

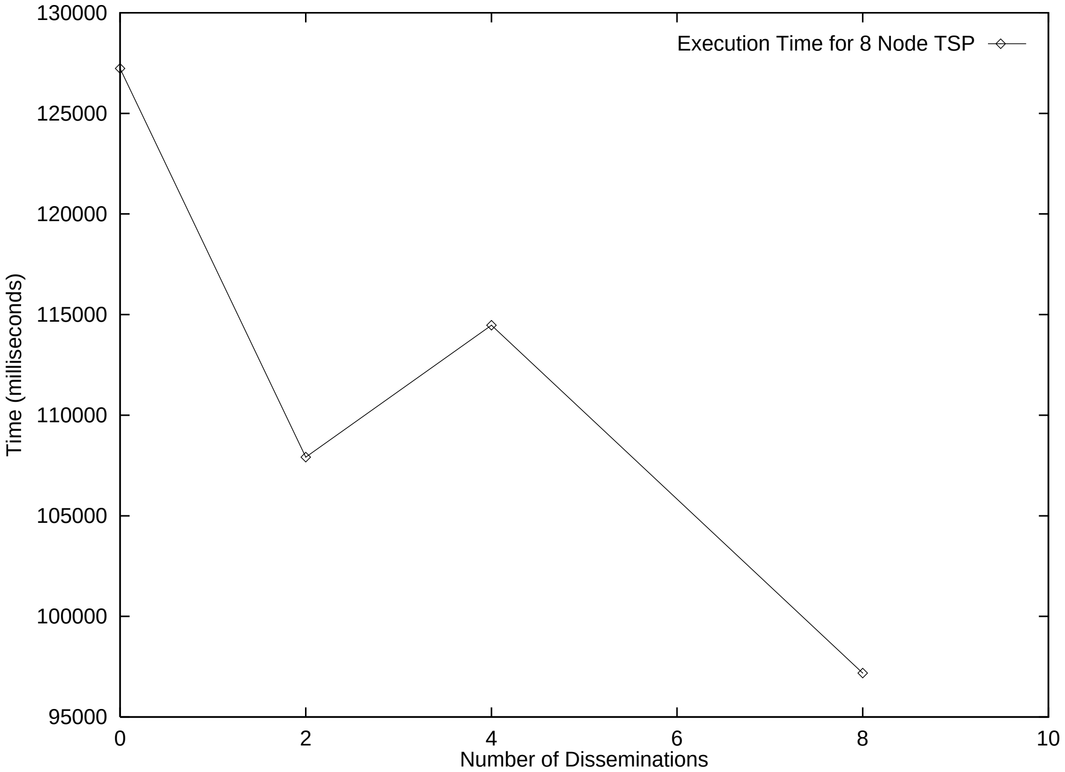

Figure 34: Completion times of 8 Node TSP (variable dissemination) in milliseconds.

distributed shared memory TSP application.

Another well known behavioral characteristic of TSP applications implemented using distributed shared memory is a decrease in completion times when an update or push-based dissemination implementation is utilized. This is due to the fact that tasks are better able to bound or "short-circuit" the investigation of a partial tour as it becomes obvious the tour has no chance of being the best tour. Figure 34 presents the completion times for the TSP problem using eight task nodes with increasing amounts of push-based dissemination. As illustrated in the Figure, for the most part, Mocha's implementation of TSP also displays this behavioral characteristic. Note that there is an anomaly when the number of tasks to receive dissemination is equal to four.

Failure Type	Execution Time	Execution - Detection Time
No Failures	97 seconds	97 seconds
Failure in 1st Half of execution	198 seconds	176 seconds
Failure in 2nd Half of execution	206 seconds	173 seconds

Table 5: Completion time of 8 node TSP with full replica dissemination (with and without failures) in seconds.

It is difficult to say exactly why this anomaly occurs because the TSP application is somewhat nondeterministic with regards to which tasks perform specific jobs. It appears however that the extra overhead required to disseminate to four tasks instead of two does not result in tighter bounding of tour investigations. Thus, completion times are not less when the number of tasks to receive dissemination is increased from two to four.

The TSP application is also useful for investigating the overheads that result when a failure occurs and the system must recover from the failure and complete execution. In the experiments that follow, we simulated the most difficult failure for Mocha to mitigate: a failure in which a node fails while holding a lock. This requires Mocha to detect the failure, break the lock, find an up-to-date copy of the replica and transfer it to the next node that desires it. Table 5 presents completion times for TSP without failures as well as with a single failure in the first half and second half of execution. Table 5 also shows completion times when the time required by Mocha to detect the failure is included or excluded from the completion time. Note that the time to detect a failure may be dictated by the timeout value set to break overdue locks. In this particular experiment the timeout value was set to 20 seconds, a value that would be appropriate for many wide area computing environments. As presented in the

Table, the overall completion time was slightly greater for a failure in the latter half of execution than a failure in the first half of execution. This anomaly was determined to be due to the synchronization thread requiring additional time to determine a lock was overdue. As the application progresses, there appears to be slightly more contention at the lock server. This is because the server is busy processing lock requests. This results in increasing the latency of detecting an overdue lock. When we remove the time associated with failure detection overhead, we obtain the more intuitive result of a failure occurring later in the execution yielding a smaller completion time. This of course is expected as a failure that occurs later in the execution has the benefit of extra computing effort from the failed node.

In order to measure the effect on performance when a failure occurs, it is useful to compare the completion times from executions in which a failure occurs, to the completion times for various number of nodes in which no failures occur. This is presented in Figure 35.

As illustrated in the Figure, the effect of a failure on an eight node TSP is to increase its completion time to approximately that of a four node execution. By plotting the completion times after subtracting out the times for failure detection, we are still left with a somewhat substantial difference in execution time. While some of this is due to failure recovery overheads as well as the reduction of computing resources, another significant contributor to the reduction in performance is redundant work that needs to be completed. This redundant work not only resulted from redoing work from a failed node but also from redoing work at the end of execution for a node that appears to be failed but in reality has not completed its last job. The master

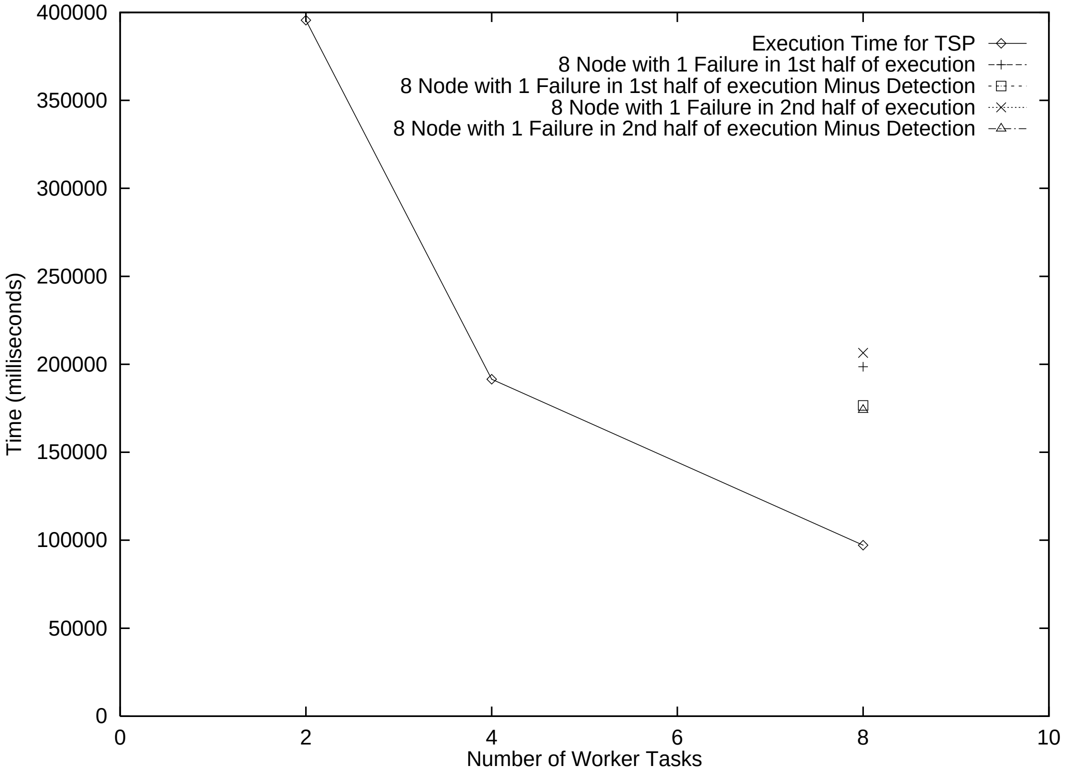

Figure 35: Execution times of TSP (with and without failures) in milliseconds.

task, after determining all jobs have been dequeued, typically encounters a will structure for a node, assumes it has failed, and reexamines the partial tour. One approach to mitigating this behavior is to have the master task wait a period of time after it determines the job queue to be exhausted. This reduces the chance of incorrectly assuming a node has failed.

5.3 CSCW Applications

Computer Supported Cooperative Work (CSCW) applications is another domain in which the functionality provided by Mocha may be beneficial. These types of applications require state sharing support to facilitate the collaborative work efforts they support. In this section, we investigate the development of two versions of a shared calendar application which utilize Mocha's mechanisms to support both a tightly coupled shared calendar as well as a loosely coupled calendar. The latter version relies on timestamps to handle update conflicts that can occur when using the looser coupling.

Figure 36 shows the graphical user interface (GUI) utilized by both versions of the shared calendar. Essentially, the calendar presents a weekly schedule where each hour has a text field associated with it that denotes an activity that is planned to occur during that hour. A text string is updated by using a mouse to press the button that is labeled with the corresponding hourly interval. This triggers a dialogue box to appear. The user may then type in the dialogue box a new message and submit the message by pressing the dialogue's button labeled *OK*. This activity results in the modification of a `StringReplica` that is associated with each text field. The `StringReplica` is a subclass of `Replica` that is generated using the `MochaGen` tool

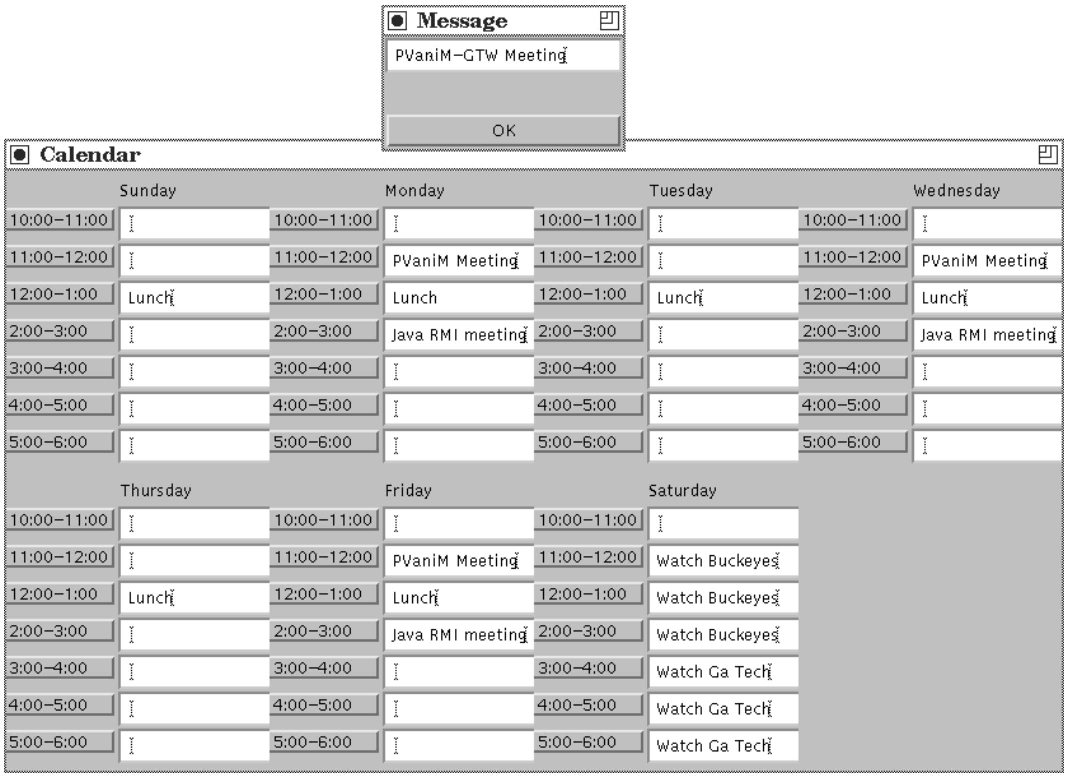

Figure 36: Mocha collaborative calendar application.

described in Chapter 4.

In the tightly coupled version of this application, there is one `ReplicaLock` with which all `StringReplica` objects are associated. The shared calendars very frequently refresh their local GUI with the newest version of the `StringReplica's` available. Thus, after very short intervals they acquire the `ReplicaLock`, utilize the new versions of the StringReplica's to update the local GUI, and then release the lock. When the dialogue box is used to update a text field, the same procedure occurs with the following difference: the `StringReplica` associated with hourly interval that is being changed is first modified to contain the text from the dialogue box before the text fields are updated from the `StringReplicas`.

In the loosely coupled version of this application, the shared calendars are permitted to perform local updates without first obtaining the most recent version of the `StringReplicas`. This of course results in the updating of a locally cached copy and can significantly reduce the overheads associated with performing an update. After relatively large intervals of time (e.g., a day), the tasks perform the process of merging the individual copies of the shared calendars. Because the policy permits each task to update only its local copy, there can exist update conflicts that have to be resolved using a policy for merging[61, 39]. In our implementation, we associate timestamps with each text field's `StringReplica` and the timestamps are updated with the current time whenever a `StringReplica` is updated. Thus we support a merging policy whereby each `StringReplica` retains the most recent update.

The loosely coupled version of the shared calendar is slightly more complicated to program than the implementation for the tightly coupled calendar. In order to implement the loosely coupled calendar, each task associates a distinct `ReplicaLock`

Loose Update	Tight Non-Cached Update	Loose Refresh Update
10.25 milliseconds	122 milliseconds	146.5 milliseconds

Table 6: Replica update times for loosely and tightly coupled shared calendars.

for its `StringReplica's`. Because each lock uses its own `ReplicaLock` when updating a `StringReplica`, the Mocha system permits the lock acquisition to occur without the need to find the most up-to-date copy of the `Replica's`. However, when an update does occur, the timestamp associated with the `StringReplica` being modified is updated with the current time of day. Now, whenever a task begins the process of synchronization (i.e., merging) of the calendars, it first makes a copy of its local `StringReplicas`. It then acquires a separate `ReplicaLock` which is used as a token to signify a task has begun the merging process. The task then iteratively acquires the `ReplicaLock's` that all the other tasks have associated with the `StringReplica's`. This allows the task to effectively pull versions of the `StringReplica's` from other tasks. These versions are then merged into the local copy of the `StringReplicas` using the timestamps to determine which version of the `StringReplica` should be considered most up-to-date and hence remain the current copy.

5.3.1 Performance

Table 6 presents performance results for both the loosely coupled and the tightly coupled shared calendar implementations. The applications execute in local area network cluster of SUN ULTRA 1 workstations. There were two tasks participating in the sharing of calendars. As shown in the table, by allowing updates to occur locally without the acquisition of the most recent version of the `StringReplicas`, the time to perform an update is reduced by a factor of twelve. However, there are some drawbacks to this approach. In addition to the fact that the calendars do not stay tightly synchronized, when a task wishes to begin the process of merging calendars and needs to pull a new version of the replicas, the time for this operation has increased by approximately 20%. This is due to the extra marshaling and transport overheads associated with sending timestamps along with the `StringReplicas`. It is also worth reemphasizing that as the number of tasks sharing calendars increases, so does the number of pull operations that must be performed during the merge process. Nonetheless, applications that need to perform a merge infrequently and do not require tight coupling can be programmed with the facilities provided by Mocha and may benefit from a loosely coupled approach.

5.4 Discussion

In this chapter we have explored several classes of applications that Mocha is able to support. The first class, electronic commerce home service applications, illustrates that Mocha is able to support an interactive electronic commerce application. This application utilizes Mocha's state sharing facilities to enable collaborative home shopping. Furthermore, we show that Mocha is capable of supporting heterogeneous environments networked to the home that are in many ways similar to what we expect to exist in the near future.

For the second class of applications, scientific applications, we illustrate how the traveling salesperson problem, a parallel branch and bound application that requires state sharing, can be developed in a failure resilient manner using the mechanisms provided by Mocha. We also provide empirical results for this application that illustrate the benefits of the dissemination capabilities provided by Mocha as well as the overheads incurred by its failure detection and recovery mechanisms.

The third and final class of applications we explore is the more traditional CSCW collaborative applications such as a shared calendar. For these applications, we illustrate that Mocha's mechanisms may be utilized to provide a variant of weakened consistency. Furthermore, we demonstrate that using Mocha in this manner can yield significant performance improvement in the cost of performing an update over collaborative applications that use Mocha's mechanisms in a tightly coupled fashion.

Chapter 6

Visualization

Understanding and analyzing the execution of a wide area distributed application can be a difficult task. Wide area computing inherits several complexities that are common to all distributed environments. Activities associated with distributed computing that may be problematic include program understanding, debugging, and performance tuning.

One approach to addressing the complexities related to distributed computing is the use of visualization. Program and/or performance visualization can serve as an aid for all of the above problematic activities associated with distributed computing[42]. Wide area distributed computing environments are no exception. The usefulness of visualization in this regard stems from (i) the highly developed image processing system possessed by humans, which allows us to track multiple complex patterns and to easily spot anomalies in these patterns; and (ii) the ability of an appropriate picture or image to convey the same amount of information as hundreds or thousands of lines of text[65]. Consequently, the textual equivalent of the information provided by a visualization may be much more difficult for a user to assimilate.

In this Chapter, several aspects of wide area computing environments which stand to benefit from visualization support are identified. These may be grouped into three categories which include wide area computing middleware execution, environmental aspects, and failure recognition. While other categories certainly exist, our experiences indicate these categories would benefit strongly from visualization support. We then describe approaches for the monitoring and visual presentation of information associated with these categories. Finally, we provide example scenarios that illustrate the benefits of visual presentation support for wide area computing environments.

6.1 Visualization Categories

Activities associated with wide area computing that are strong candidates for visualization support include the operation of wide area computing middleware, environmental aspects, and failure recognition.

6.1.1 Wide Area Middleware Operation

Wide area middleware operation refers to activities performed by the system infrastructure during the execution of a wide area application. Prominent examples of middleware operation include remote evaluation and state sharing support. Remote evaluation may benefit from visualization support due to the inherent complexity of the activity as well as its potential for being a bottleneck. *Mocha Servers*, which provide the class files requested by a task, can become a bottleneck if several remote tasks are concurrently requesting class files. Recall that the Mocha system relies upon two techniques for mitigating remote evaluation related bottlenecks. First, the caching of classes which have already been remotely fetched is used to avoid redundant dynamic network loading transfers. Second, the use of dynamic network loading is short-circuited whenever it can be discerned that the class that must be fetched is a portion of the Mocha library or the standard Java library. Visualization of these activities can provide insight into the efficacy of these approaches for reducing bottlenecks. Additionally, graphical views can depict the amount of filespace being utilized for caching activities. For remote sites where filespace is limited, this may identify applications which utilize too much filespace and therefore are inappropriate for execution at certain remote sites.

Another core component of this activity which has an impact on performance is the amount of communication latency that results from dynamic network loading. Should communication latency appear to be substantially high, the visual presentation of this information to the user helps to identify a potential performance bottleneck.

Graphical views that provide information about state sharing activities may also yield valuable information regarding middleware operation. State sharing in the Mocha system is a relatively complex procedure which involves lock acquisition and release, and the marshaling and transferring of shared data. For example, standard object serialization marshaling for certain classes may be prohibitively expensive, and a graphical view of this may motivate an application developer to create custom marshaling code which provides better performance. Moreover, lock acquisition and release may be too fine grained and result in a bottleneck at the synchronization thread which may be mitigated by restructuring the application or executing the synchronization thread on a more efficient workstation.

6.1.2 Environmental Aspects

Wide area computing systems execute in network computing environments. Typically, applications execute on workstations or PCs that have varying capabilities and configurations in terms of CPU speed, memory, local vs. networked disks, etc. Furthermore, in many cases, each computer as well as the network itself, is potentially subject to uncontrollable external loads. These factors often result in load imbalances and dynamic fluctuations in delivered resources which can be a major cause of performance degradation[53]. Visualization systems that provide information regarding the environment in which a distributed application executes can provide valuable insight into how these environmental factors impact performance[62].

Several aspects of the environment in which Mocha wide area distributed applications execute are ideal candidates for visual presentation. The amount of external load present on workstations is one such example. Once high external loads on workstations are recognized, this workstation may be avoided by future tasks that will execute at remote sites. In select circumstances, it may be possible that personal requests (via email) may be made to have the loads on remote resources reduced.

Another aspect of the environment suitable for presentation is the amount of memory being utilized by the tasks on the remote resources. Should a task begin to use a significant portion of the memory available, it may be necessary to restart the application on a remote node that has more memory to avoid thrashing and/or memory exhaustion.

A final aspect of wide area computing environments that may benefit from visualization is the latency and reliability of the network connecting the remote tasks. Connections with high latency or above normal packet loss rates should be presented such that they may be quickly recognized visually.

6.1.3 Failure Recognition

Depicting where failures occur in a wide area computing application is also a strong candidate for visual presentation. Failures can have a significant impact on the performance and operation of a wide area distributed application. Informing application developers where failures have occurred can serve as an aid to understanding how failures have affected the execution of the application. Furthermore, historical profiles of where failures have been common in the past may identify hosts that are undesirable due to their weak reliability.

6.2 Monitoring Wide Area Computing Activities

Visualization is dependent on some form of monitoring to record interesting aspects of a wide area computing application. These records may then be interpreted by a visualization tool to produce a graphical presentation.

Monitors utilize two fundamental techniques for information collection: tracing and sampling[48]. In tracing, all occurrences of events are stored for a certain interval of time. Typically, this interval is for the duration of the distributed application. In sampling, occurrences of an event are collected asynchronously, typically only at the request of the monitor.

Tracing utilizes sensors which are small pieces of code that are embedded within the program and perform the desired recording of information. Although complex techniques for developing sensors exist[48], simple techniques are surprisingly useful. For example, many distributed systems[58, 9] supply library routines for communication, synchronization, and spawning of tasks. These integral events are traced by providing macro wrappers that first perform the tracing operation and then call the desired routines.

Sampling may be performed by sensors or in some cases by probes, which reside in the monitor and directly access the address space of the application[48]. Sampling is useful when one may only need cumulative statistics such as the number of sends and receives by a node at various stages of the execution of an application. Utilizing probes (when possible) can minimize the perturbation to the application that would be incurred had sensors been utilized.

The monitoring of wide area applications can be problematic due to the high communication latency that exists between the cooperating tasks. This can have a significant impact on the amount of on-line monitoring that may be performed. Whereas an on-line monitoring system for a multiprocessor can use the machine's high performance I/O channel and extra monitoring threads to allow fine grain on-line tracing with acceptable monitoring delays[28], a wide area computing environment with higher communication latencies and comparatively less bandwidth will be unable to support this type of functionality.

Similar problems have been faced by monitoring systems that provide visualizations for cluster computing environments[62]. These problems are addressed by such systems by adopting a two phase approach, with run time or *on-line* sampling based monitoring focusing on the types of visualization that are mandatory for use during execution, and postmortem trace event based monitoring being relegated to detailed program analysis and tuning. This two phase approach is a natural result of attempting to adapt traditional multiprocessor and multicomputer visualization techniques for use in network computing environments. A major benefit of this two phase approach is that it allows for on-line visualizations that render at speeds that are much closer to the real-time execution of the application, as well as being able to generate trace data useful in subsequent postmortem examinations[62].

Monitoring a wide area computing environment is possible by tailoring the two phase approach developed for cluster computing environments to take into account the additional limitations imposed by wide area computing environments. The limitations include low bandwidth communication and disk space limitations that may exist at remote hosts. Low bandwidth may be mitigated by reducing the on-line sampling rates and having the graphical views inform the user when it is presenting information that is lagging substantially behind the real-time execution of the application. Disk space concerns may be addressed by severely limiting the amount of event tracing that is performed. In our design of a visualization subsystem for wide area computing environments, we conservatively assume that no disk space is available for event tracing and thus present a visual presentation scheme that consists solely of on-line graphical views which utilize sampling based monitoring.

6.3 Visual Presentation

Developing a suitable visual presentation of activities associated with wide area computing can be difficult due the large amount of information to be communicated. In addition to needing information from all tasks executing in order to obtain some global context regarding the execution of the application, in some situations it is necessary to focus on specific details regarding the activities of an individual host or even an individual thread which executes on the host. This wide range of visual presentation needs can be difficult to satisfy.

One possible solution to this problem is through the use of a tiled multilevel browser[50] to coordinate the graphical presentation of wide area computing activities. With this approach, views are provided at varying levels of detail and typically include a global view, an intermediate view, as well as more detailed views. The application developer is continuously provided with the notion of global context thus providing an overview of the activities occurring. Simultaneously, the intermediate and detail views may be used to focus in on specific aspects of the application. Figure 37 provides an illustration of a multilevel browser display that could be used for wide area computing activities. At the top of the Figure is a global view which provides an overview of the wide area computing application. In this view, a rectangle is provided which serves as a magnifier and is used to delineate the boundaries of the intermediate view shown at the bottom left of Figure 37. The rectangle may be positioned anywhere in the global view and dragged to any location using an interaction device such as a mouse. To the right of the intermediate view in Figure 37 is a detail view which provides more specific information about the intermediate view.

Figure 38 provides an example of how a global view might visually provide information regarding the activities of a Mocha application. In this view, *Mocha Servers*, which encapsulate the execution of a thread in the system are represented as circles. For most applications, we expect an upper bound for the number of *Mocha Servers* utilized to be approximately a hundred. We expect a global view to be able to present a majority of the *Mocha Servers* without requiring excessive scrolling of

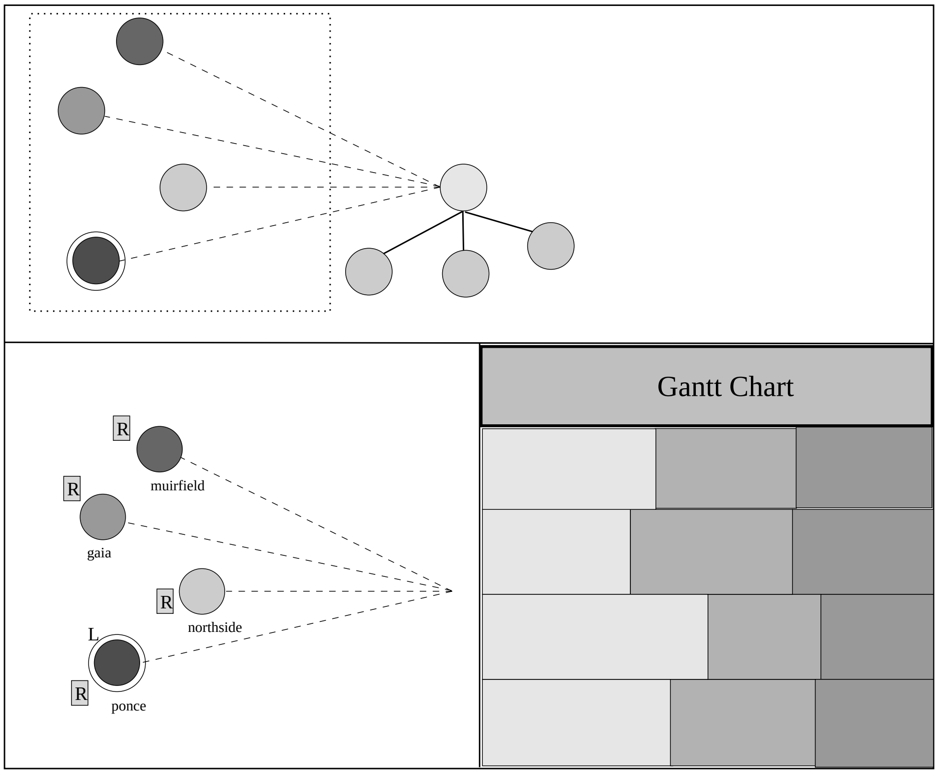

Figure 37: Illustration of a multilevel browser for wide area computing activities.

the view. The circles are positioned in clusters relative to the communication subnet in which the *Mocha Servers* they represent are located. Providing an aesthetic presentation of a graph such as this has been the focus of a substantial amount of research. Graph layout algorithms developed to generate high-quality drawings quickly enough for interactive use have been implemented[22] and are well suited to the layout requirements of the Mocha global view.

Colors are used to illustrate certain traits about the *Mocha Servers*. For example, lighter colors indicate that system overheads are within some (user defined) acceptable range while darker colors indicate bottlenecks. Furthermore, a round robin scheme may be utilized such that the circles are colored at various time intervals to represent

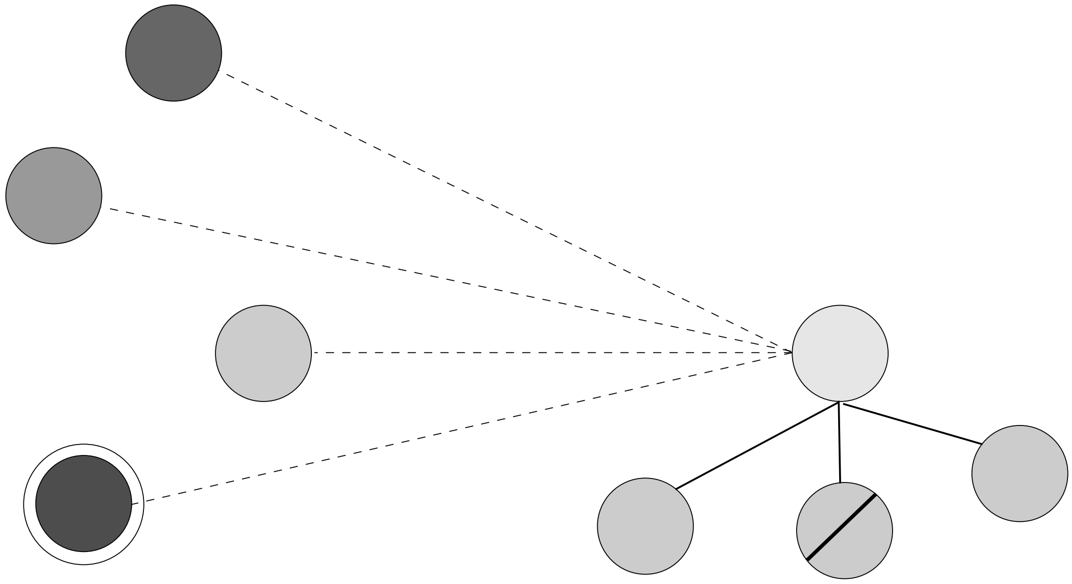

Figure 38: Example global view for Mocha.

different characteristics. In this scenario, for a few seconds the color will represent remote evaluation overheads and then for the next few seconds the color may refer to state sharing overheads such as the time to transfer replicas. At other intervals, loads on the host, the amount of class file caching performed, or memory use might be presented. It is also possible for the user to take control of the round robin scheme and designate which characteristic is currently represented by the circle. Lingering problems that require immediate in-depth investigation are denoted with an outer ring that encompasses the circle. MochaServers that have been detected as failing (via timeouts) have a line drawn through the center of the circle.

Packet loss is represented in this view by the type of line that connects the two *Mocha Servers*. A dotted line indicates that packet drops are numerous whereas a solid, darker line indicates a low packet drop rate. At select intervals, the lines may also serve to represent the communication latency between *Mocha Servers*.

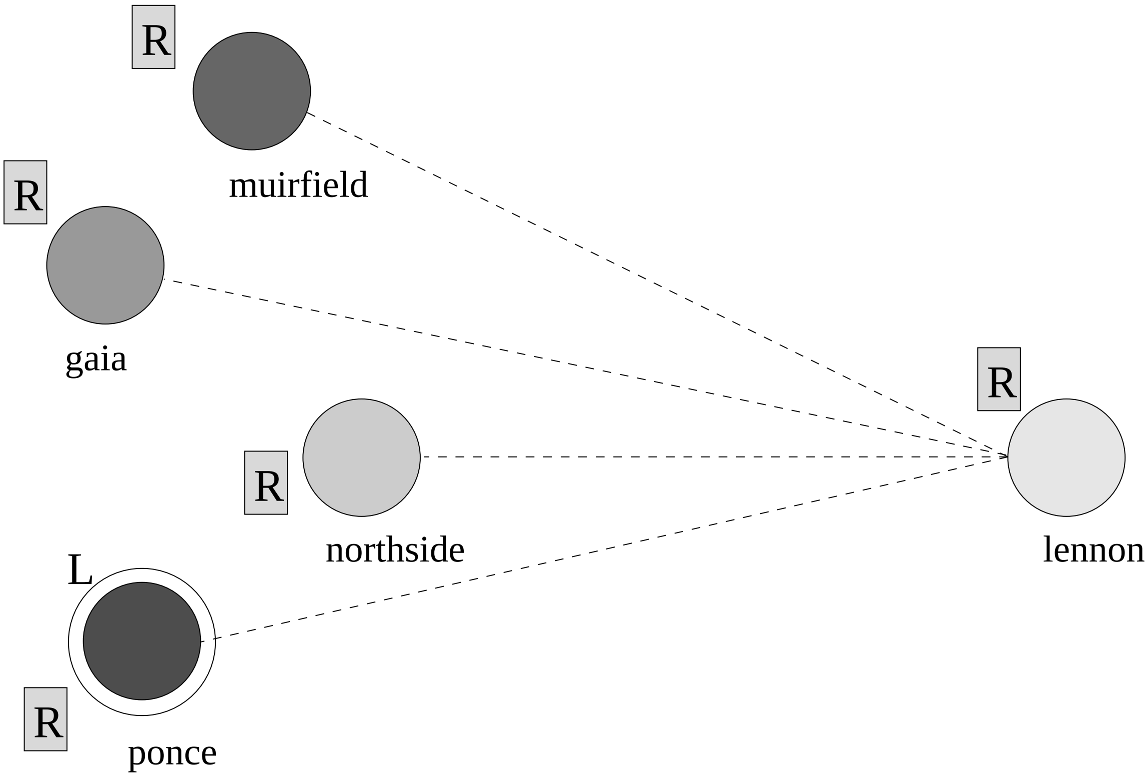

Figure 39: Example of an intermediate view for Mocha.

Figure 39 provides an example of an intermediate view for the global view presented in Figure 38. Essentially, this view provides a zoomed or magnified view of the global view and thus has the capability to present more information. For example, this view provides the host names on which the *Mocha Servers* execute. It also represents the replicas located at each *Mocha Server* as rectangular icons. Lock ownership is denoted by placing an **L** adjacent to the circle representing a *Mocha Server* that contains a thread that has acquired the lock.

The intermediate view permits the user to interactively select the graphical objects located in the view for presentation in the detail view. The detail view allows for an in-depth investigation of a single or small group of *Mocha Servers*. Detail views useful

Wide Area Computing Aspect	Visual Presentation
Lock Acquisition Overheads	Gantt Chart
Lock Release Overheads	Gantt Chart
Marshaling Overheads	Gantt Chart
Data Transfer Overheads	Gantt Chart
Remote Evaluation Overheads	Gantt Chart
Memory Utilized on Remote Hosts	Bar Chart
Task Load on Remote Hosts	Bar Chart
Packet Drops (Mocha NCL)	Bar Chart
Number of Failures	Bar Chart
Number of Shared State Transfers	Bar Chart
Remote Host Disk Storage Utilized	Bar Chart
Application Output	Textual View
Stack Dumps / Exception Conditions	Textual View
List of Remotely Evaluated Class Files Cached	Textual View

Table 7: Mapping of wide area computing aspects to detail visual presentation techniques.

for presenting information about a *Mocha Server* include Gantt charts, bar charts, and textual displays. Table 7 provides a mapping of wide area computing aspects to detail visual presentation techniques.

Many wide area computing activities listed in Table 7 may be presented using a Gantt chart detail view. A Gantt chart view is a bar chart variant in which task activities are shown graphically as bars against the horizontal X-axis and different colors are used to represent different activities. Typically, a horizontal row of bars depicts the activities of a single thread and the activities of multiple threads may be depicted by grouping multiple rows of bars together. Figure 40 is an example of this type of view. In this Figure, a Gantt chart is provided for one of the *Mocha Servers* which illustrates the overall performance of the state sharing activities being

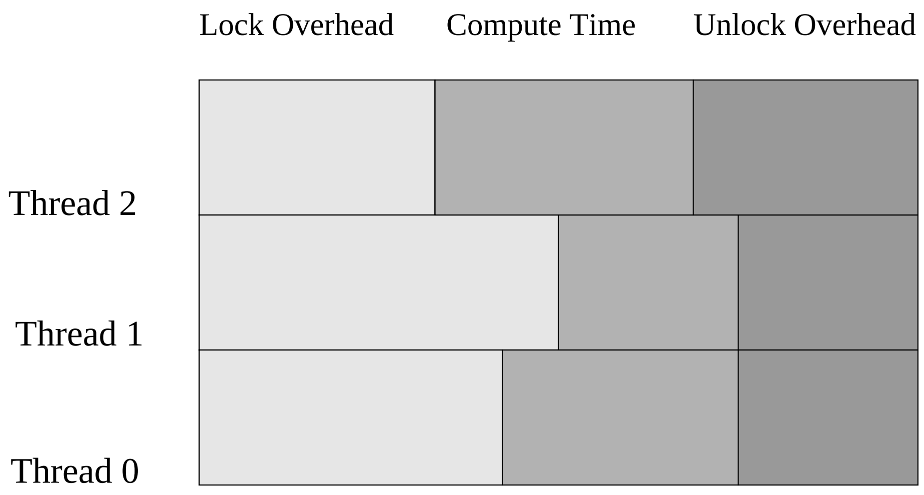

Figure 40: Example of an Gantt chart detail view for Mocha.

performed by the threads in this *Mocha Server*. Here three colors are used to represent the time performing lock acquisition, the time the thread computes and the time for lock release. This view permits the performance of individual threads to be presented. When the number of threads executing in a *Mocha Server* is relatively small, the threads from a group of *Mocha Servers* may be presented simultaneously. Gantt charts are also useful for illustrating the amount of overhead that results from class file transfers required for remote evaluation of a task.

Another informative detail view is the bar chart. Bar charts provide a graphical means of comparing numerical values by mapping these values to rectangles whose lengths are proportional to the numbers represented. Bar charts are valuable for visually presenting several aspects of wide area computing. For example, bar charts are well suited for presenting the process load or memory usage for a group of remote hosts. Comparing the relative number of packet drops by Mocha's NCL at various hosts is also well illustrated by bar charts. Other activities that map well to this presentational methodology include the number of failures that have occurred at a host over a given time period and the amount of disk storage being utilized at remote hosts.

Textual views are also useful as detail views. In particular, textual views effectively display information that is not easily amenable to graphical presentation. Examples include output from tasks that result from the use of Mocha's remote `println()` capabilities as well as the remote printing of stack dumps when errors occur. A listing of the class files that have been cached at remote sites is also well suited for textual presentation and can provide insight into caching needs of applications.

In order to support the detail views above, an obvious problem that must be resolved is how to make the system information that is required to "drive" the above views actually available and accessible to a visualization subsystem. One possible solution to this problem is to integrate support for visualization directly into a distributed system. In this approach, system routines are modified directly to generate trace events that contain system-specific information needed by the visualization system. Additionally, communication and synchronization mechanisms are modified to transfer visualization related information between tasks. An example of this is the "piggybacking" of Lamport clock values onto system communication and synchronization messages which is necessary to permit individual tasks to maintain an accurate Lamport clock for use in timestamping trace events. The availability of this timestamp in event traces greatly simplifies the task of creating software visualizations that accurately depict the execution of a distributed application[63, 64].

Another approach is to utilize environment probes to acquire information about remote hosts[64]. In this approach, a thread is instantiated on the remote host whose sole responsibility is to obtain pertinent information regarding the status of the remote host. For example, on a Unix workstation, an environment probe would utilize system calls such as `getrusage()` to obtain resource utilization information such as memory usage and leverage off of Unix commands such as `uptime` to provide load average information. The environment probe would then communicate this information to the visualization subsystem. More information regarding the use of these techniques for visualizing distributed systems may be found in [62, 63, 64].

6.4 Examples

In this section, we provide two example scenarios in which visualization support provides insight into the execution of a wide area computing application. Suppose there exists a wide area computing application whose execution produces the visualization presented in Figure 41. Here, the global view illustrates through the use of dashed lines containing large separations between the line segments that the network communication infrastructure is dropping an unusually large number of packets. Upon closer inspection with the intermediate and Gantt detail view, it becomes evident that this is having a severe impact on the application's performance. In particular, the Gantt view shows that lock acquisition and release overheads are an extremely large fraction of the overall execution time and only a small amount of time is being spent performing computation on behalf of the application. In this situation, it is highly recommended that the user search for other remote hosts that currently have more reliable network connectivity. It is worth noting that this type of information could also be presented textually. However, as the number of hosts increases, presenting the above information in textual screens becomes problematic due to the difficulty

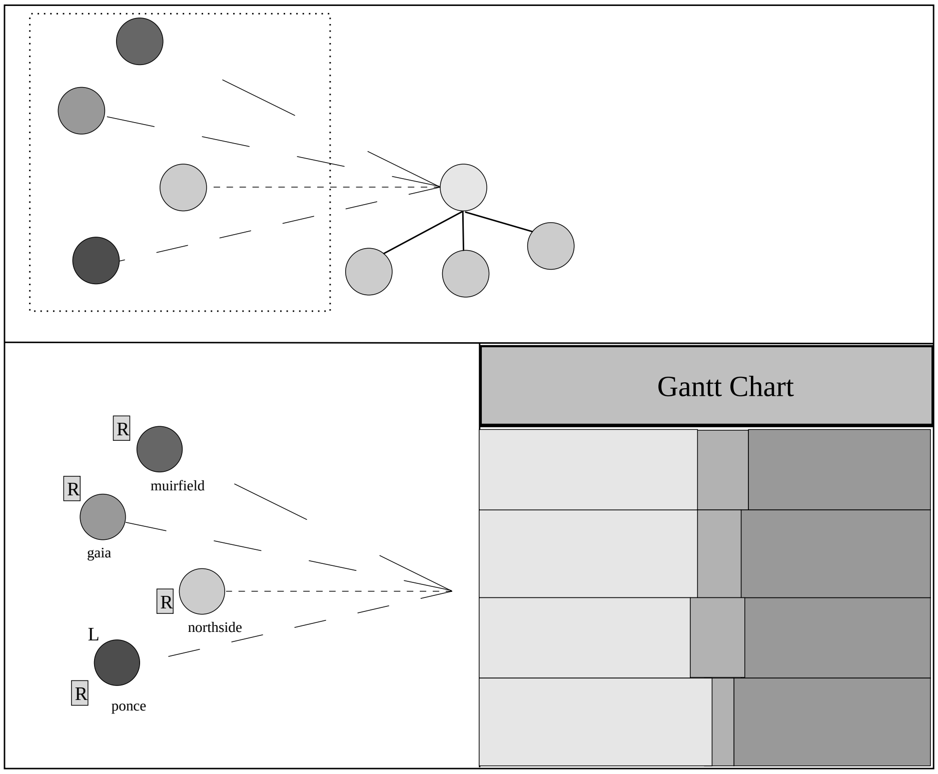

Figure 41: Example usage scenario of wide area computing visualization.

involved with continuously reading and comprehending the large streams of text that must be generated to provide the equivalent amount of information provided in the graphical displays.

Let's assume the user has found other remote hosts that are suitable for the wide area computing application. Suppose this execution generates the visualization illustrated in Figure 42. In this scenario, an anomaly at a remote site has been recognized and presented to the user in the form of a ring which surrounds the circle representing the *homepark* remote host. Upon closer inspection via the intermediate and detail views, it becomes evident that the host has a large number of jobs in its run queue and thus has a highly loaded CPU. In this situation, if the application

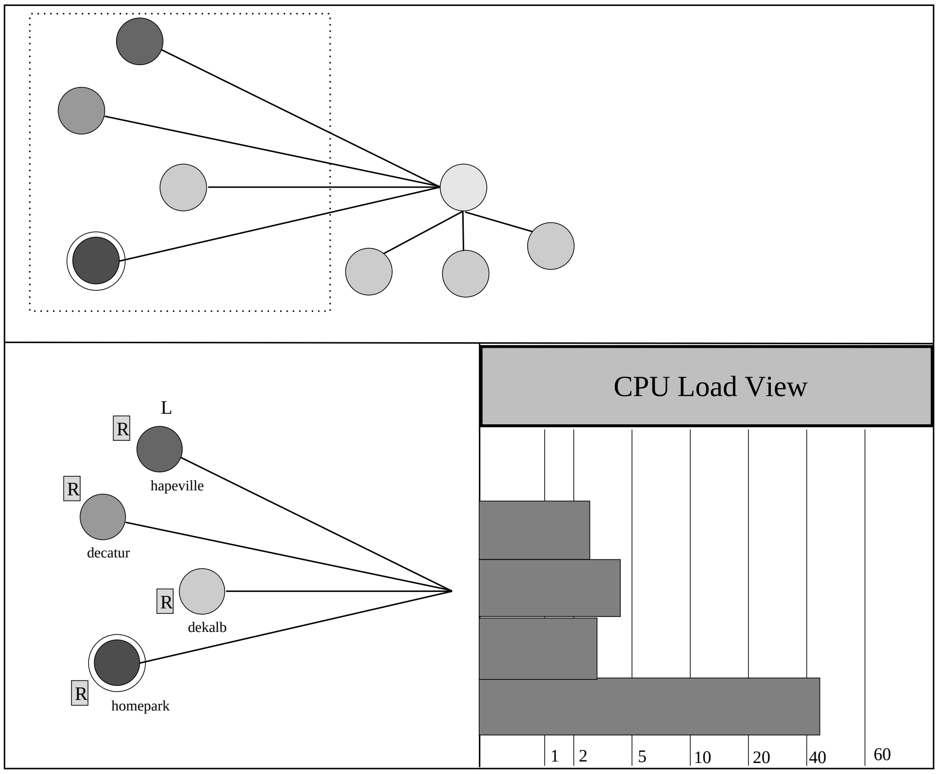

Figure 42: Example usage scenario of wide area computing visualization.

has been written using an inherent dynamic load balancing strategy, the application can proceed without interruption. Because the visualization illustrates that a large majority of the remote hosts are lightly loaded, and a dynamic load balancing strategy is being utilized, it is assumed the application will be able to adjust to the unusually high load present on *homepark*. If the application is not using some form of dynamic load balancing, the user should terminate the application and find a substitute for the overloaded host.

6.4.1 Conclusions

The graphical views and examples presented in this chapter are intended to illustrate situations in which visualization support would provide insight into wide area computing environments. The selected methods of visual presentation are based on experiences gained from building wide area computing middleware and applications. It is worth noting, however, that visualization systems benefit from the feedback acquired from actual users of a system. Thus, this chapter has served as a starting point as well as an impetus to other researchers who may be interested in performing a more empirical evaluation of the use of visualization in wide area computing environments.

Chapter 7

Conclusions and Future Work

Wide area computing domains such as metacomputing and home service environments are poised to become a reality. Recent advances in Internet computing tools such as Java as well as better network connectivity to the home are factors that contribute to wide area computing networks becoming a popular target environment for research in distributed computing. In many cases, wide area distributed computing domains require more advanced capabilities than those provided by a standard web browser. Such capabilities must address issues such as heterogeneity, object sharing, and failure resilience in wide area environments.

The principal contribution of this thesis is the design and implementation of Mocha, a Java based middleware infrastructure system to enable wide area applications. The middleware provides mechanisms to support wide area computing including remote evaluation support, support for failure resilient shared objects, hybrid communication protocol support, debugging and tools support, and platform independence. Additionally, we have performed a detailed empirical evaluation of the system in local area, wide area, and home service networks, and we have developed design strategies for providing visualization support for wide area computing.

Mocha's remote evaluation support permits applications to be spawned at remote sites quickly. Furthermore, these applications execute in a secure "sandbox" fashion that limits the application's ability to act maliciously at the remote site. In addition, we use the notion of a *Site Manager* that is responsible for controlling the number of true processes on the remote site that are allocated for use by foreign tasks.

Support for interactivity has been provided in the form of shared objects. Mocha provides support for shared objects on heterogeneous platforms. To improve performance and mitigate communication latencies, copies of objects can be created and accessed locally. The object sharing support utilizes advanced distributed shared memory techniques for maintaining consistency of shared objects. Moreover, failure resilience support that allows its overhead to be controlled based on the level of availability needed by an application has been integrated into the shared object model.

In order to better support the communication requirements of object sharing, we have investigated the design of a "multiple protocol" approach that combines the capabilities of Mocha's own lightweight network library and the TCP protocol to support efficient transfer of object replicas in a scalable fashion.

Tool support has been provided that simplifies the process of deploying an application. In addition, debugging support has been developed that automatically ships debugging information from a remote site back to the root site at which the application began execution. This enables troubleshooting to be performed without contacting a user located at the remote site. Additionally, we have investigated design strategies for developing visualization support for wide area computing that aid in the understanding and performance tuning of wide area applications.

The key observation of this thesis is that wide area computing middleware that provides the mechanisms discussed above is able to support several classes of wide area computing applications including electronic commerce home service applications, scientific applications, and traditional computer supported cooperative work applications. Furthermore, these applications can be created and deployed quickly in a heterogeneous environment in a failure resilient fashion.

7.1 Future Work

There are several open problems associated with this thesis work that are excellent avenues for future work. These include experimenting with wireless networks, investigating the impact of techniques that improve the performance of Java, exploring non-synchronization based shared object consistency models, performing rigorous scalability studies, implementing Mocha's design strategies for visualization, and integrating support for charging for remote resources.

It would be very interesting to evaluate Mocha's capabilities in wireless networks. Wireless networks are a substantially different environment than the local area, wide area, and home service networks that we have already used to empirically evaluate Mocha. It is an open question how well Mocha's mechanisms would fare in this domain.

Another interesting issue not yet explored is the effect an optimized implementation of Java (e.g., JIT compilation) would have on the performance of Mocha's protocol support. Currently, Mocha's network communication library is best suited for small control messages used by the Mocha system. It is possible that when user level Java code becomes more efficient, the network communication library may be able to perform fragmentation and reassembly more efficiently and thus be useful for larger sized messages. A related issue is of course to determine the performance loss for a scientific parallel computing application using this optimized Java version of Mocha compared to writing the application with traditional concurrent network computing substrates such as PVM.

Mocha's consistency actions are driven by synchronization operations, even when the mechanisms are used in a weakly consistent fashion. In certain systems and applications, synchronization for all operations may not be desirable. For example, systems such as Bayou[61], Coda[39], and Rover[35] that address mobility avoid synchronization and instead rely upon conflict detection and resolution to maintain consistency. Exploring non-synchronization based consistency models that are suitable for supporting shared objects is another potential avenue for future work.

While Mocha has been designed for scalability with regards to its protocol support and consistency maintenance mechanisms, a scalability evaluation of the system has not been performed. A thorough scalability evaluation which investigates Mocha's ability to scale with regards to large numbers of users as well as very large communication latencies would undoubtably provide substantial insight.

Mocha's design strategies for visualization support for wide area computing are based on experiences gained from building wide area computing middleware and applications. Because visualization systems benefit greatly from the feedback acquired from actual users, developing an actual implementation of the visualization system would enable a more thorough evaluation and thus is an avenue for future work.

Finally, another open question is how to integrate support for charging for remote resources. With the vast amount of computing resources now made available via the Internet and infrastructures such as Mocha, it is possible that sites' may wish to charge fees for those who wish to execute applications on the sites. For this to be possible, fee assessment models as well as techniques for preventing fraud must be developed.

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak ordering–a new definition. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in orca. In *Proceedings of the International Conference on Computer Languages*, pages 79–89, 1990.

[3] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. ATLAS: An infrastructure for global computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 165–172, Connemara, Ireland, September 1996.

[4] R. Baraglia, M. Cosso, D. Laforenza, , and M. Nicosia. Integrating PVaniM into WAMM for monitoring meta-applications. In M. Bubal, J. Dongarra, and J. Wasniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 226–233. Springer-Verlag Lecture Notes in Computer Science LNCS 1332, November 1997.

[5] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[6] Krishna A. Bharat and Luca Cardelli. Migratory applications. In *Proceedings of UIST '95*, pages 133–142, Pittsburgh, Pa, November 1995.

[7] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[8] A. D. Birrell and J. Nelson. Implementing remote procedure calls. *ACM TOCS*, 2(1):39–59, Feb. 1984.

[9] Doug Bowman, Adam Ferrari, Brian Schmidt, Melisa Schmidt, Brad Topol, and Vaidy Sunderam. The Conch network concurrent programming system. Technical Report CSTR-940301, Emory University, Atlanta, GA, March 1994.

[10] Tim Brecht, Sandhu Harjinder, Meijuan Shan, and Jimmy Talbott. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 181–188, Connemara, Ireland, September 1996.

[11] Henri Casanova and Jack Dongarra. NetSolve: A network server for solving computational science problems. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996. To appear.

[12] K. Mani Chandy et al. A world-wide distributed system using Java and the Internet. In *Fifth IEEE International Symposium on High-Performance Distributed Computing (HPDC-5)*, Syracuse, NY, August 1996.

[13] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

[14] David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *Proc. Twelfth ACM Symposium on Operating Systems, Operating systems Review*, 23(5):202–210, December 1989.

[15] P. Ciancarini and R. Tolksdorf. Using the web to coordinate distributed applications. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[16] Michael Condict, Dejan Milojicic, Franklin Reynolds, and Don Bolinger. Towards a world-wide civilization of objects. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 25–32, Connemara, Ireland, September 1996.

[17] Symantec Corporation. Just-in-time compiler for windows 95/NT, 1996.

[18] Samir R. Das, Richard M. Fujimoto, and John T. Stasko. Animating the execution of time warp programs. Technical Report GIT-CC TR-91/35, Georgia Institute of Technology, August 1991.

[19] Keith Edwards. *Collaborative Infrastructure in Collaborative Systems.* PhD thesis, Georgia Institute of Technology, 1995.

[20] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steve Tuecke. Multimethod communication for high-performance metacomputing applications. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996. To appear.

[21] Geoffrey Fox et al. WebWork: Integrated programming environment tools for national and grand challenges. Technical Report NPAC SCCS-715, Syracuse Univesity, Syracuse, NY, June 1995.

[22] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[23] G.A. Geist et al. PICL:A Portable Instrumented Communication Library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Lab., Oak Ridge, Tenn., 1990.

[24] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Seventeenth International Symposium on Architecture*, pages 2–14, May 1990.

[25] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, May 1990.

[26] Paul Gray and Vaidy Sunderam. IceT: Distributed computing and Java. In *Proceedings of ACM 1997 Workshop on Java for Science and Engineering*, Las Vegas, Nevada, June 1997.

[27] Andrew S Grimshaw and William A. Wulf. Legion flexible support for wide-area computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 205–212, Connemara, Ireland, September 1996.

[28] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, and Jeffrey Vetter. Falcon: On-line monitoring and steering of large-scale parallel programs. Technical Report GIT-CC-94-21, Georgia Institute of Technology, Atlanta, GA, 1994.

[29] M. Heath, A. Malony, and D. Rover. Parallel performance visualization: From practice to theory. *IEEE Parallel and Distributed Technology*, pages 44–60, November 1995.

[30] M. Heath, A. Malony, and D. Rover. The visual display of parallel performance data. *IEEE Computer*, pages 21–28, November 1995.

[31] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29 –39, September 1991.

[32] David P. Helmbold, Charlie E. McDowell, and Jian-Zhong Wang. Traceviewer: A graphical browser for trace analysis. Technical Report UCSC-CRL-90-59, Univ. of California at Santa Cruz, Santa Cruz, CA, October 1990.

[33] Dag Johansen, Robert van Renesse, and Fred B. Schneider. Supporting broad internet access to TACOMA. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 55–58, Connemara, Ireland, September 1996.

[34] Ranjit John and Mustaque Ahamad. Evaluation of causal memory for data-race-free programs. Technical Report GIT-CC-94/34, College of Computing, Georgia Tech., 1994.

[35] Anthony D. Joseph et al. Rover: A toolkit for mobile information access. In *Proc. Fifteenth ACM Symposium on Operating Systems*, pages 156–171, Copper Mountain Resort, CO, dec 1995.

[36] http://www.javasoft.com/people/richb/jsda/, 1997.

[37] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency. In *Proceedings of the 19th Annual International Symposium On Computer Architecture*, pages 13–21, 1992.

[38] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium of Computer Architecture*, 1992.

[39] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computing Systems*, 10:3–25, 1992.

[40] Prince Kohli, Mustaque Ahamad, and Karsten Schwan. Indigo: User-level support for building distributed shared abstractions. *Concurrency: Practice & Experience*, 10(1):1–29, January 1998.

[41] David Kotz, Robert Gray, and Daniela Rus. Transportable agents support worldwide applications. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 41–48, Connemara, Ireland, September 1996.

[42] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, June 1993.

[43] Ted Lehr, David Black, Zary Segall, and Dalibor Vrsalovic. Visualizing system behavior. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II–117—II–123, August 1991.

[44] Laura Lemay and Charles Perkins. *Teach Yourself Java in 21 Days*. Sams.net, Indianapolis, Indiana, 1996.

[45] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4):321–359, November 1989.

[46] Allen D. Malony, David H. Hammerslag, and David J. Jablownski. Traceview: A trace visualization tool. *IEEE Software*, 8(5):29–38, September 1991.

[47] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report CS-94-230, Computer Science Dept., University of Tennessee, April 1994. Also appears in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994.

[48] David M. Ogle, Karsten Schwan, and Richard Snodgrass. The dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[49] James E. Pitkow and Colleen M. Kehoe. Emerging trends in the WWW user population. *Communications of the ACM*, 39(6), 1996.

[50] Catherine Plaisant, David Carr, and Ben Shneiderman. Image-browser taxonomy and guidelines for designers. *IEEE Software*, 12(2):21–32, March 1995.

[51] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 18th International Conference on Parallel Processing*, pages 160–169, August 1989.

[52] R. Riggs et al. Pickling state in Java. In *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 241–250, Toronto, Ontario, June 1996.

[53] Brian K. Schmidt and Vaidy S. Sunderam. Empirical analysis of overheads in cluster environments. *Concurrency: Practice & Experience*, 6(1):1–33, February 1994.

[54] David Socha, Mary L. Bailey, and David Notkin. Voyeur: Graphical views of parallel programs. *SIGPLAN Notices*, 24(1):206–215, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[55] Softway. Introduction to guava, 1996.

[56] John Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[57] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[58] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[59] V.S. Sunderam and Vernon J. Rego. EcliPSe: A system for high performance concurrent simulation. *Software: Practice & Experience*, 21(11):1289–1219, November 1991.

[60] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[61] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 1994 Symposium on Parallel and Distributed Information Systems*, September 1994.

[62] Brad Topol, John T. Stasko, and Vaidy Sunderam. PVaniM: A tool for visualization in network computing environments. *Concurrency: Practice & Experience*, 1997. To appear.

[63] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. Integrating visualization support into distributed computing systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 19–26, Vancouver, B.C., May 1995.

[64] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. The dual timestamping methodology for visualizing distributed application behavior. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems (Euro-PDS '97)*, pages 81–86, Barcelona, Spain, June 1997.

[65] Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990.

[66] David J. Wetherall and David L. Tennenhouse. The ACTIVE IP option. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 33–40, Connemara, Ireland, September 1996.

[67] Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. In *Proceedings of ACM 1997 Workshop on Java for Science and Engineering*, Las Vegas, Nevada, June 1997.

Vita

Brad was born in Melbourne, Florida on March 26, 1971. He attended Emory University from 1989 to 1993. While at Emory, he was invited to participate in Emory's Honors program as well as its accelerated Masters program. He earned both his Bachelors and Masters degree in Mathematics/Computer Science from Emory University in May, 1993. Brad graduated Summa Cum Laude and was selected to become a member of Phi Beta Kappa.

Brad decided to pursue a Ph.D. in Computer Science, and from 1993 to 1998, he attended the Georgia Institute of Technology. He specialized in distributed computing systems and received his Ph.D. in Computer Science in June, 1998.

Aside from doing research, Brad enjoys, tennis, weight-lifting, and traveling.

At this time, Brad is currently looking forward to a career in industry where he plans on doing what he enjoys most; the design and implementation of distributed systems and applications that are both useful and usable.

A Framework for the Development of

Wide Area Distributed Applications

Brad Byer Topol

146 Pages

Directed by Professors Mustaque Ahamad and John Stasko

The growth in popularity of the World Wide Web has resulted in the development of a new generation of tools tailored to Internet computing activities. Prominent examples include the Java programming language and Java capable Web browsers. These Web spinoffs are having a profound impact on the field of distributed computing. Whereas distributed computing has traditionally focused on improving the functionality of local clusters of computers, technology is progressing such that wide area computing networks are now becoming a popular target environment for research in distributed computing.

With wide area distributed computing environments, geographically distributed resources such as workstations, personal computers, supercomputers, graphic rendering engines, and scientific instruments will be available for use in a seamless fashion by parallel applications. Many envision that it will be possible to transport application code to remote sites in the wide area virtual computer where it may be executed in the presence of needed resources. The area of research devoted to bringing this vision to reality in the context of scientific applications is referred to as metacomputing.

Metacomputing environments are useful for a variety of distributed and parallel applications, particularly those which need access to remote resources or applications that are able to effectively utilize a substantial number of computing resources that the Internet may easily provide. Moreover, other wide area distributed computing

domains such as electronic commerce home service applications require more advanced capabilities than those provided by a standard web browser. Such capabilities must address issues such as heterogeneity, object sharing, and failure resilience in wide area environments.

In this thesis, we investigate the design and implementation of a Java based middleware infrastructure system to enable wide area applications. We then provide an empirical evaluation of our prototype system for local area, wide area, and home service network environments. We also illustrate the system's ability to support the development of several classes of application domains which include electronic commerce home service applications, failure resilient scientific applications, and traditional computer supported cooperative work applications. Finally, we present a design for the monitoring and visual presentation of activities associated with wide area distributed computing.