JMAS: A JAVA-BASED MOBILE ACTOR SYSTEM FOR HETEROGENEOUS DISTRIBUTED PARALLEL COMPUTING

By

LEGAND L. BURGE III

Bachelor of Science
Langston University
Langston, Oklahoma
1992

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1995

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 1998

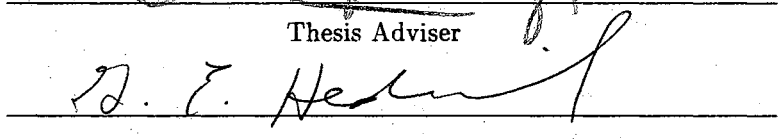
JMAS: A JAVA-BASED MOBILE ACTOR SYSTEM

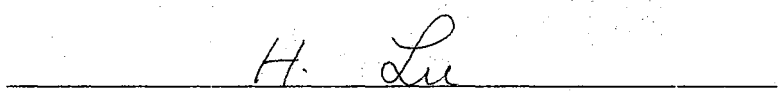
FOR HETEROGENEOUS DISTRIBUTED

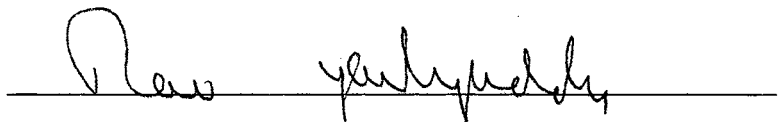
PARALLEL COMPUTING

Thesis Approved:

Thesis Adviser







Dean of the Graduate College

ACKNOWLEDGMENTS

I sincerely thank my graduate adviser Dr. K. M. George for the guidance, help and time he has given me for the completion of my thesis work. His direction and leadership helped inspire me to venture into the advanced aspects of this work. Without the encouragement and help he has given me, the completion of this work would have been impossible. I also sincerely thank Dr. H. Lu, Dr. George Hedrick, and Dr. Rao Yarlagadda for serving on my committee. Their suggestions have helped me to improve the quality of this work.

My special thanks goes to Dr. In Hai Ro and President Dr. Ernest L. Holloway, from Langston University, for the support that they have giving me throughout my studies here at Oklahoma State University.

My respectful thanks goes to my parents Dr. Legand L. Burge Jr. and Gwenetta V. Burge for all the love and support they have given me in my life. I also would like to thank all other members of my family for the love, encouragement and confidence they have endorsed in me. Finally and foremost, I thank God for giving me the opportunity to pursue a dream, and for the blessing of all who made that dream come true.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Multicomputers represent the most promising developments in computer architecture due to their economic cost and scalability. With the creation of faster digital high bandwidth integrated networks, heterogeneous multicomputers are becoming an appealing vehicle for parallel computing, redefining the concept of supercomputing [Tan92, Sta84]. As these high bandwidth connections become available, they shrink distances and change our models of computation, storage, and interaction. With the exponential growth of the World Wide Web (*WWW*), the web can be used to exploit global resources, such as CPU cycles, making them available to every user on the Internet [BL96, Rey97]. The combined resources of millions of computers on the Internet can be harnessed to form a powerful global computing infrastructure consisting of workstations, PCs, supercomputers, and computing devices such as WebTelevision (Figure 1.1).

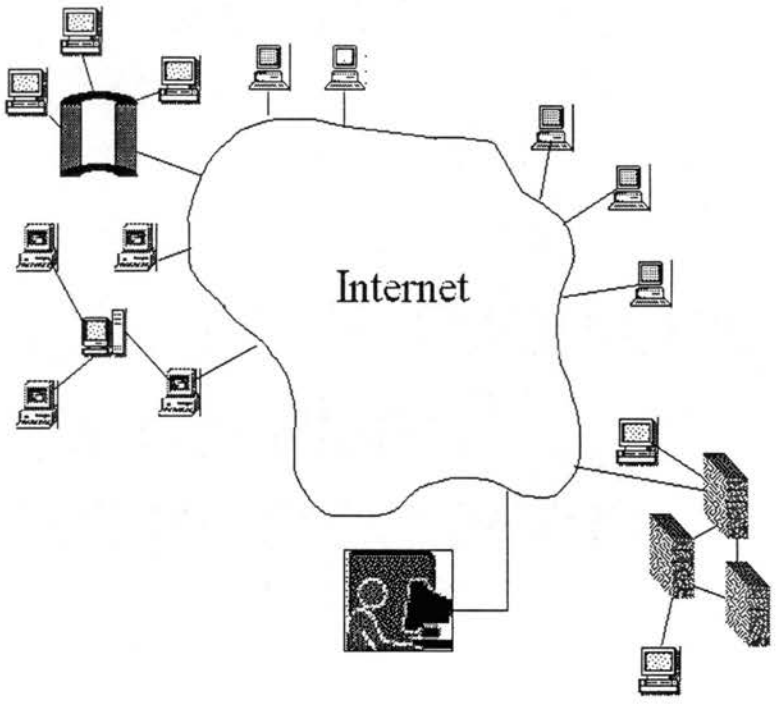


Figure 1.1 Global Computing Infrastructure.

1.1 Global Computing

The vision of integrating network computers into a global computing resource is as old as the Internet[BL96][Sta84][GWtLT97]. Such a system should hide the underlying physical infrastructure from users and from programmers, provide a secure environment for resource owners and users, support access and location of large integrated objects, be fault tolerant, and scale to millions of autonomous hosts. Some recent network computing approaches include CONDOR [LL88], MPI [GLS94], PVM [Sun90], Piranha [GK92], MIST [SaOGI97], NEXUS [LI97], Network of Workstations (*NOW*) [ACP95], Legion [GWtLT97], GLOBUS [FK97], and WebOS [Rey97]. The Message Passing Interface (*MPI*) is a standard intended for use by all those who want to write portable message passing programs in Fortran 77 and C. The MPI interface is suitable for use by general MIMD programs, as well as those written in the more restricted style of SIMD. Parallel Virtual Machine (*PVM*) is a programming environment for the development and execution of large concurrent or parallel applications. It permits a heterogeneous collection of UNIX computers hooked together by a network to be used as a single large parallel computer. The major goal of the Legion project is to provide secure shared object and namespaces, application-controlled fault-tolerance, improved response time, and greater throughput. Multiple language support is another goal. CONDOR is a software package for executing computation intensive type jobs on UNIX workstations connected by a network. NEXUS is a portable run-time system for task-parallel programming languages. It supports multiple threads of control, dynamic processor acquisition, dynamic address space creation, a global memory, and asynchronous events. GLOBUS is viewed as a networked virtual supercomputer also known as a metacomputer: an execution environment in which high-speed networks are used to connect supercomputers, databases, scientific instruments, and advanced display devices. The project aims to build a substrate of low-level services such as: communication, resource location and scheduling, authentication, and data access, on which higher-level metacomputing software can be built. MIST combines migratable PVM with global scheduling and load monitoring. MIST is designed to use idle cycles scavenged from shared networks of workstations to run existing PVM

programs efficiently and effectively. WebOS is being developed with the objective of providing operating system services for wide area applications such as resource discovery and management, a global namespace, remote process execution, authentication, and security. The goal of Piranha is to provide adaptive parallelism. Adaptive parallelism refers to parallel computations on a dynamically changing set of processors: processors may join or withdraw from the computation as it proceeds. Networks of fast workstations are the most important setting for adaptive parallelism. Workstations at most sites are typically idle for significant fractions of the day, and those idle cycles may constitute a powerful computing resource. Most of these systems require the user to have login access to all machines used in the computation. In order to achieve heterogeneity all systems require the maintenance of binaries for all architectures used in the computations.

1.2 High-Performance Computing With Java

With the recently released standard Java components such as Remote Method Invocation (*RMI*) [atoSM96b], Object Serialization [atoSM96a], Java IDL (interface to the CORBA domain), and performance boosters such as JIT Java compilers, the HPCC community has been rapidly producing a collection of Java APIs in the following areas to support high-performance computing:

- matrix algebra
- image primitives
- PDE primitives
- parallel compiler primitives (e.g. lexers, parsers)
- performance visualization and monitoring
- load balancing resource allocation, and cluster management

Recently, researchers have proposed several approaches to provide a platform independent Java-based high-performance global computing infrastructure. These include Javalin [DoCS96a, DoCS97c],

Java/DSM [DoCS97b], WebFlow [aSU97b, FF96b, aSU97a, DoCS97c], IceT [oMCS97], JavaDC [DoCS97a], Parallel Java [KBW97], Parallel Java Agents [KAB98], ATLAS [BBB96], Charlotte [BKKW96], ParaWeb [BSST96], Popcorn [CLNR97], and Ninflet [TMN98]. The use of Java as a means for building distributed systems that execute throughout the Internet has also been recently proposed by Chandy et al. [CDL+96], Fox et al. [FF96a] and implemented in [Van97, Ven97]. Javalin is an infrastructure for global computing. The system is based on Internet software that is interoperable, increasingly secure, and ubiquitous. Javalin's architecture and implementation require participants to have access to only a Java-enabled Web browser. Javalin is a prototype system that consists of brokers, clients, and hosts. WebFlow is a particular programming paradigm implemented over WebVM and follows a dataflow programming model. WebVM is a mesh of servers that manage and coordinate distributed computations. A WebFlow application is given by a computational graph, visually edited by end-users using Java applets. The aim of the IceT project has been to mutually incorporate approaches and techniques found in Internet programming with established and evolving distributed computing paradigms. It is a novel framework for collaborative and high-performance distributed computing which is built upon a Java substrate. IceT addresses the ideas of harnessing geographically-remote resources for anonymous utilization, portability of processes and data, and distributed security issues. JavaDC - Java for Distributed Computing - is a web-based environment for managing the execution of parallel and distributed SIMD applications written using MPI [GLS94], and PVM [Sun90] and does not provide support for most distributed heterogeneous computing applications. Parallel Java Agents is a framework for parallel computing in locally confined scalable computing clusters based on agents that communicate through asynchronous messages. Charlotte supports distributed shared memory, and uses a fork-join model for parallel programming. Tasks may be submitted to several servers, providing fault-tolerance and ensuring timely execution. ParaWeb provides two separate implementations of a global computing infrastructure, each with a different programming model. Their Java Parallel Class Library implementation provides new Java classes that provide a message-passing framework for spawning threads on remote machines and sending and receiving messages. ParaWeb's Java Parallel Runtime System

is implemented by modifying the Java interpreter to provide global shared memory and to allow transparent instantiations of threads on remote machines. ATLAS provides a global computing model, based on Java that is best suited for tree-based computations. ATLAS ensures scalability using a hierarchy of managers. POPCORN provides a Java API for writing parallel programs. POPCORN applications are decomposed by the programmer into small, self-contained subcomputations, called *compulets*. The API facilitates something like RMI, except the POPCORN application does not specify a destination on which the compulet is to run the execution. Rather, a "market" which brings together buyers and sellers of the CPU, determine which seller will run the compulet. Ninflet facilitates RPC based computing of numerical task in a wide area network. Java/DSM is a platform for programming heterogeneous environments using Java and software Distributed Shared Memory (*DSM*).

1.3 Mobile Agent Technology

Mobile agents are a convenient paradigm for distributed computing [Doc95, DoCS96b, Whi94b, Whi94a, Inc94, Gra95, BFD96, Rie94, SH97, UoCaI96, Pei97, LDD96, KKT92, oCS96, Age97, rL96, HBB96, MC96, oT96, Age97]. The agent specifies when and where to migrate, and the system handles the transmission. This makes mobile agents easier to use than low-level facilities in which the programmer must explicitly handle communication, but more flexible and powerful than schemes such as process migration in which the system decides when to move a program based on a small set of fixed criteria. A mobile agent carries all of its internal state with it which eliminates the need for separate communication steps. The agent migrates to a machine performs a task, migrates to a new machine, performs a task that might be dependent on the outcome of the previous task and so on. Mobile agents allow a distributed application to be written as a single program. Mobile agents can be viewed as extensions of the client/server model. Clients and servers can program each other and applications can dynamically distribute its server components when it starts execution.

Early mobile agents systems include: Agent Tcl [Gra95], HTTP-based mobile agents [LDD96], Messengers[UoCaI96], Obliq[fDRC94], Telescript [Whi94b], TACOMA [oT96], and Ara [Pei97].

Agent Tcl is a prototype system that uses the Tool Command Language (*TCL*) as the mobile agent language. Agent Tcl is provided with explicit commands to send/receive Tcl scripts to/from remote machines. The infrastructure for HTTP-based mobile agents uses the Hypertext Transfer Protocol (*HTTP*) for agent transfer and communication, taking advantage of this widely accepted, platform independent protocol. Agents are encapsulated in MIME-like messages for transport. The TACOMA project focuses on operating system support for mobile agents and how agents can be used to solve problems that are traditionally done by operating systems. TACOMA is based on UNIX and TCP. The system currently supports the following languages: C, Tcl/Tk, Perl, Python, and Scheme. Ara - Agents for Remote Actions - is an application independent and language-neutral execution platform for mobile agents written in general interpreted languages. Obliq is an object-oriented scripting language for distributed computation. Computations are defined in terms of Oblets; active objects that are distributed over multiple machines. Telescript technology is similar to Agent Tcl, with the exception of more security features imposed in the system. It is used to supplement system programming languages such as C and C++. Messengers are autonomous objects [BFD96], each capable of navigating through the underlying virtual network and performing various tasks at each node. Applications are written from the point of view of the Messenger as they navigate through the system. Applications therefore can compute in unknown network topologies. Applications that have been suggested for mobile agents include distributed information retrieval, active documents, active e-mail, network management, electronic commerce, controlling remote devices, and collaborative applications.

Some recent mobile agent systems provide support for platform independent Java applications. These include: ObjectSpaces Voyager [Obj96], General Magics Odyssey [Obj96], and IBM's Aglets [rL96]. All of the current systems are 100 percent pure Java and use features of JDK 1.1 [DD96, Inc95]. With the release of mobile agent systems developed using the new standard Java components, as well as, performance boosters such as JIT Java compilers, Java-based high-performance global computing may soon become a reality.

1.4 Thesis

In this thesis we introduce *mobile actors*; a parallel programming paradigm for distributed parallel computing based on mobile agents and the *actor* message passing model [AHP91, BVN91, Hew77]. The Actor-based message passing model supports dynamic architecture topologies that make it ideal for distributed parallel computing. We implement a prototype system (*JMAS*) based on the mobile actor model using Java technology [DD96, Ham96, Inc95]. In particular, we provide a mobile actor API (*Application Programming Interface*) for writing mobile actor programs. Applications are decomposed by the programmer into small, self-contained subcomputations and distributed among a virtual network of Distributed Run-Time Managers (*D-RTM*); which execute and manage all mobile computations. Lastly, we evaluate the performance of our system, and show that our system is well suited for course grain computations in a global heterogeneous environment. Our experiments were ran using two benchmarks: a Mersenne Prime Application , and the Traveling Salesman Problem. In summary, this thesis is three-fold:

1. To introduce the mobile actor programming paradigm for global computing based on mobile agents and the actor message passing model.

2. To implement a prototype system based on mobile actors using Java technology.

3. To evaluate the performance of our mobile actor system.

1.5 Organization

The thesis is divided into the following chapters:

- Chapter 2: A detailed review of literature related to current Java-based global computing architectures that motivate the research conducted in this thesis is presented.

- Chapter 3: Discusses thesis objectives and theoretical foundations on which this research is based. In particular, we introduce mobile actors.

- Chapter 4: Discusses technical issues associated with the configuration of a Mobile Actor System. Introduces JMAS: A Java-Based Mobile Actor System.

- Chapter 5: Discuss the implementation specific design of the JMAS architecture. In particular, we discuss the design of the Distributed Run-Time System and how Java technology is applied.

- Chapter 6: Discusses the performance evaluation of our mobile actor system.

- Chapter 7: A summary of the thesis and suggestions for future work are presented.

- Appendix A: Contains a detailed installation/users guide for JMAS.

- Appendix B: Gives the specification for the JMAS Mobile Actor API.

- Appendix C: Includes JMAS example programs.

CHAPTER 2

LITERATURE REVIEW

There has been much work in providing collaborative use of computational resources over a global network. As described in Section 1.1, some models of global computing use low-level communication systems, others use high-level dedicated systems. Although these systems offer heterogeneous collaboration of multiple systems in parallel, they involve rather complex maintenance of different binary codes, multiple execution environments, and complex underlying architectures. Distributed computing over networks (i.e. local networks, intranets, or the internet), has emerged as a technology with tremendous promise and potential, owing in part to the emergence of the Java Programming Language and the World Wide Web. Java, because of its platform independence, overcomes the complexity issues of maintaining different binary codes, multiple execution environments, and complex underlying architectures. It offers the basic infrastructure needed to integrate computers connected to the Internet into a distributed computational resource for running parallel applications on numerous anonymous machines.

In the following sections, we give a detailed description of several Java-based high-performance global computing infrastructures that motivate the research conducted in this thesis. The systems can be largely categorized into two – those that use *active objects* [BFD96] [SH97] [CM96] [fDRC94] as a computing paradigm [DoCS96a][DoCS97a] [TMN98] [BBB96] [oMCS97] [CLNR97] [BKKW96][aSU97b], and those which do not [BSST96][DoCS97b] [KAB98].

2.1 Ice T

The aim of the IceT project [oMCS97] has been to incorporate approaches and techniques found in Internet programming with established and evolving distributed computing paradigms. This includes the ideas of harnessing geographically-remote resources for anonymous utilization, portability of processes and data, as well as, security issues that may arise. Extended features incorporated into IceT include: dynamic merging and splitting of virtual machines, multi-user awareness, portability

of processes and data across multiple virtual machines, and a framework for multi-user programs. IceT builds upon traditional message-passing paradigms found in distributed computing, such as PVM [Sun90], and combines with it wide accessibility and portability of processes as found in both Java and Internet programming paradigms. Processes in IceT execute in parallel across multiple networks, among multiple users, and share information using IceT's message-passing substrate. Unlike PVM, processes need not be provided to each computational resource in the IceT environment. Processes are able to transfer and install themselves on remote machines providing unbounded file system and network access. The major Java components that make up IceT consist of a ClassLoader class, SecurityManager class, and the Java Virtual Machine (*JVM*). Through the Java ClassLoader, processes represented as Java bytecode can be uploaded, instantiated, and executed on remote hosts. Programmers must explicitly supply the address of the code to be uploaded. Bytecode executing on remote hosts has no login or file access and is subject to security restrictions imposed by the owner of the hosts. The SecurityManager class provides a means for imposing elementary security restrictions. The JVM provides Java bytecodes with a uniform, system-independent view of the underlying architecture.

2.2 JavaDC and ARCADE

JavaDC and ARCADE [DoCS97a] are developed with the premises that web is becoming an attractive framework for solving distributed applications. This is realized in particular because a web interface can be made platform independent. The ARCADE environment extends the focus of JavaDC to more general applications which consists of multiple heterogeneous modules interacting with each other to solve an overall problem.

2.2.1 JavaDC

JavaDC - Java for Distributed Computing - is a web-based environment for managing the execution of parallel and distributed SIMD (*Single Instruction Multiple Data*) applications written using MPI [GLS94] and PVM [Sun90]. Such applications based on this model execute the same program on different subsets of the program data. JavaDC does not provide support for more distributed

heterogeneous computing applications. Users of JavaDC are able to develop a parallel environment on a high-performance workstation cluster (*HPC*) in one domain, run an application on the HPC using executables and input/output files located in another domain, and monitor its progress. As shown in Figure 2.1, the architecture of JavaDC consists of four components:

- The *Web Client* is a WWW browser. The *Java Client* is a graphical user interface which runs under the browser environment. The Java Client interacts with the user and communicates with the *Java Server* over the network specifying the application and resource requirements for the target systems.

- The *Web Server* is a HTTP-server. The Java Server is a Java-based server which is the main working engine of JavaDC. It collects the users specification and executes the application on the HPC.

- The HPC (*High Performance Cluster*), is a cluster of workstations which form the execution environment.

- The application site stores the application executables and data input/output.

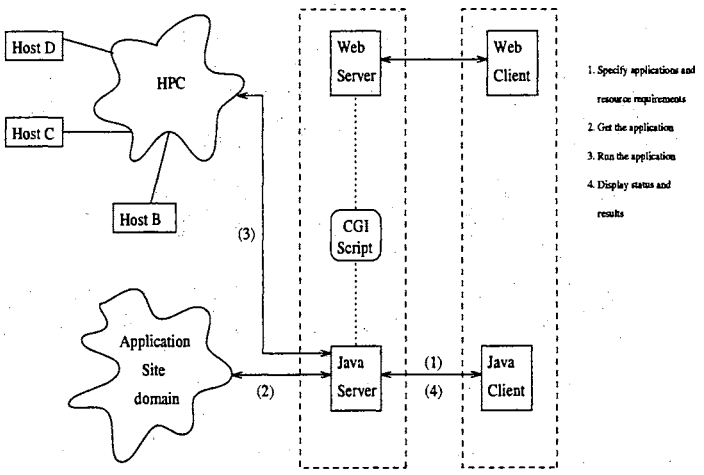


Figure 2.1 Architecture of JavaDC taken from [DoCS97a].

2.2.2 ARCADE

ARCADE is an integrated environment designed to provide support to collaboratively design, execute, and monitor multidisciplinary applications on a distributed heterogeneous network of workstations [DoCS97a]. It extends JavaDC to more general applications. The system consists of the following sub-environments:

- The *monitoring and steering interface*, allows multiple users to monitor the execution status of the applications, as well as, examine intermediate and final results of large data sets.

- The *resource allocation and execution interface*, provides support for specifying the hardware resources required for the execution of the application. Resources could be chosen statically or dynamically during the execution by the users or by the system based on the current and predicted loads of the systems.

- The *application design interface*, allows the hierarchical specification of the execution modules and their dependencies. (similar to dataflow)

The overall goal of the environment is to provide an easy to use, portable, and easily accessible environment that provides support through all phases of application development and execution. As shown in Figure 2.2, the ARCADE system is a three-tier architecture. The front-end consists of a web browser (client), which interacts with the Java User Interface Server (*UIS*) that provides information and services as needed by the client. The UIS also launches an Execution Controller (*EC*), which manages the overall execution of the application by starting up user modules on the specified resources. This is a centralized approach to managing and executing user computations on HPCs. The EC interacts with the last-tier called the Process Controllers (*PCs*), which run on individual resources in the execution environment. All of the logic is embodied in the middle tier, allowing the front-end and back-end to be very thin, thus making it feasible to run on low-end machines and keep additional loads on the execution machines to a minimum.

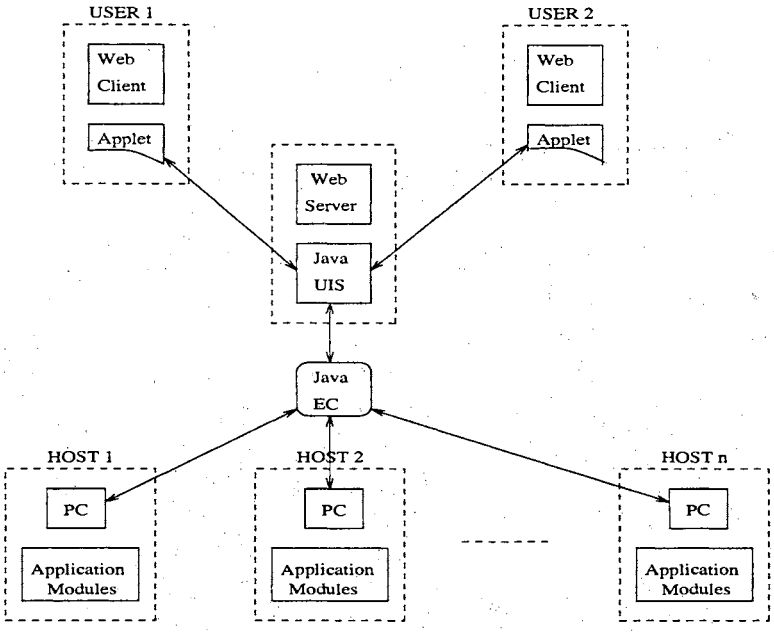


Figure 2.2 Architecture of ARCADE System taken from [DoCS97a].

2.3 Java/DSM

Common parallel programming paradigms consist of shared memory (for multiprocessors), and message passing (for multicomputers). Shared memory provides the advantage of a convenient message depository for fast processor to processor communication. Such systems are complex and are very hard to build. In converse, multicomputer systems are easy to build with the disadvantage of a more complex software system. Efforts to combine the advantages of multiprocessors (easy to program) and multicomputers (easy to build) have lead to communication paradigms that simulate shared memory on multicomputer systems. These paradigms allow multicomputers to communicate through Distributed Shared Memory (*DSM*) [BN96]. Distributed Shared Memory is an attractive abstraction because it provides processes with uniform access to local and remote information. This uniformity of access simplifies programming, eliminating the need for separate mechanisms to access local state and remote state information. Java/DSM is a system for programming heterogeneous computing environments based upon Java and software DSM [DoCS97b]. Java/DSM transparently handles the hardware differences and the distributed nature of the system. The underlying software

system is complex and requires costly communication strategies to maintain memory coherence. Java/DSM consists of a JDK-1.0.2 based parallel Java Virtual Machine on the TreadMarks DSM system [KCDZ94]. It includes a distributed garbage collector and supports the Java API with very few changes.

2.4 WebFlow

WebFlow [aSU97b, FF96b] is a general purpose Web based visual interactive programming environment for coarse-grain distributed computing built on top of standards such as HTTP, Html, and Java [LDD96] [Ham96]. The environment consists of a three-tier architecture with the central control and integration WebVM layer in tier-2, interacting with the visual graph editor applets in tier-1 and legacy systems in tier-3. The WebVM layer (tier-2) is a runtime environment that consists of a mesh of Java web servers such as Jeeves [Inc97] from Javasoft or Jigsaw [Con96] from MIT/W3C which coordinate distributed computation represented as a set of interconnected coarse-grain dataflow Java modules. Modules run asynchronously, are mobile and communicate by exchanging Java objects along their dataflow channels. WebFlow is a particular programming paradigm implemented over WebVM and given by a dataflow programming model. Modules are written by developers who need not be concerned with management and coordination functions. As shown in Figure 2.3, WebFlow management is implemented as three sub-components: SessionManager, ModuleManager, and ConnectionManager. The SessionManager receives graph specifications from the graph editor applet (tier-1), and creates an image of the whole compute-web using module proxy objects called ModuleRepresentations. WebFlow then decides on the decomposition strategy and notifies the ModuleManager to start a ModuleWrapper that runs modules and notifies the ConnectionManager about the connectivity required by a module.

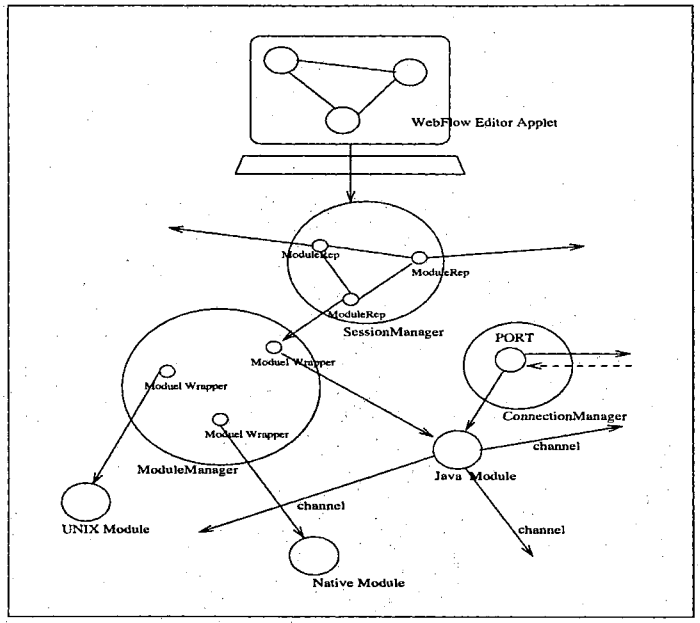


Figure 2.3 Design of WebFlow Management taken from [aSU97b].

2.5 Javalin

Javalin is a Java-based global computing infrastructure that consists of three kinds of participating entities: *clients*, *brokers*, and *hosts* [DoCS96a]. A client is a process seeking computing resources; a host is a process offering computing resources; and a broker is a process that coordinates the supply and demand for computing resources. As shown in Figure 2.4, Javalin is a simple architecture which enables anyone connected to the Internet or and Intranet (via a Web browser) to participate. Clients register with brokers their tasks to be run, hosts register with brokers their intentions to run tasks. Brokers assign the tasks to the host and forward the results back to the client. By simply pointing a hosts browser to a known broker URL, users automatically make their resources available in a distributed computation. In particular, Javalin software is downloaded to the hosts browser. This software allows the host to accept tasks to be executed in favor of the broker.

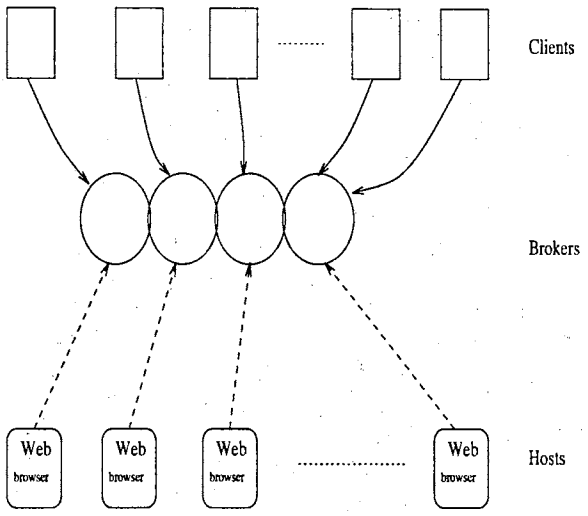


Figure 2.4 Javalin Architecture taken from [DoCS96a].

2.6 ParaWeb

Within the ParaWeb framework, users may execute existing unmodified thread-based Java applications remotely on faster computation resources, or they may execute them in parallel on a variety of platforms across the Internet [BSST96]. ParaWeb achieves parallelism in Java through two approaches:

1. A Java Parallel Runtime System (*JPRS*).

2. A Java Parallel Class Library (*JPCL*).

The JPRS support a distributed shared memory framework, whereas, the JPCL supports message passing. Executing standard thread-based program across multiple platforms on the JPRS requires presenting these threads with the illusion of shared memory. Providing communication mechanisms between remote threads supported only by the JPCL class library is easiest to do using a message passing framework. The parallel class library in ParaWeb provides mechanisms for the remote creation and execution of threads, and facilitates communication between them. As shown in ParaWeb's architecture (Figure 2.5), the following sequence of steps are performed in order to execute a parallel program.

1. Each server (i.e. A, B, and C) start daemons which register with a load scheduling server (which keeps track of idle servers).

2. Remote threads are created when the client contacts the scheduling server to request an address of a remote idle server.

3. The scheduling server responds with the address of the requested server.

4. The client sends compiled byte-code to the remote server for execution.

5. The remote server returns the results.

6. The client then notifies the scheduling server to inform that it has finished using the remote server.

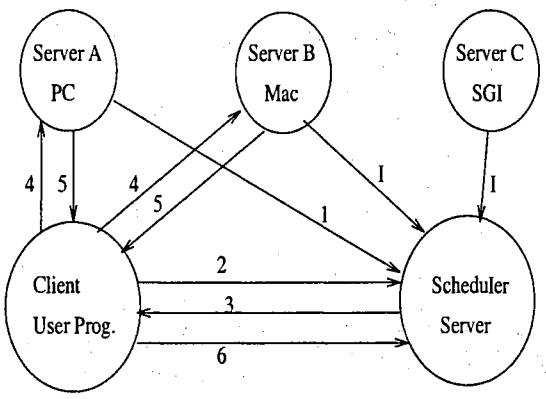


Figure 2.5 Architecture of ParaWeb taken from [BSST96].

The JPRS is a modified Java interpreter which allows threads to be instantiated on any remote machine that is running the modified interpreter. The Java interpreter on each machine coordinates with the other interpreters to maintain the illusion of a global shared address space. Consistency among replicas in the global shared address space is enforced through a release consistency protocol. The runtime system relies on existing Java mechanisms for concurrency and synchronization.

2.7 ATLAS

The ATLAS system is designed to exploit network resources of the world as a giant distributed computer, and to develop an infrastructure that exploits idle resources [BBB96]. ATLAS realizes this by combining existing mechanisms and policies from Java and Cilk [BJK+95] together with some new mechanisms and policies that extend ATLAS into a global computing infrastructure. Cilk is a C-based parallel multithreaded programming language together with a runtime system that provides thread scheduling based on the technique of work stealing (i.e. idle processors steal threads from random machines). Cilk also provides fault-tolerance and adaptive parallelism. ATLAS adapts the Cilk programming model to Java (to enforce heterogeneity); it extends the Cilk work-stealing scheduler; and it borrows mechanisms to provide adaptive parallelism and fault tolerance. The ATLAS system architecture consists of clients, managers, and compute servers (Figure 2.6).

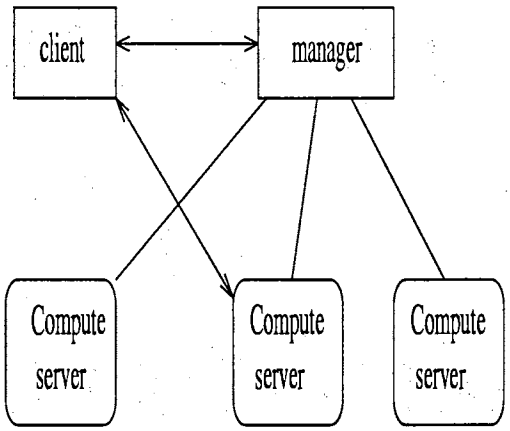


Figure 2.6 Architecture of ATLAS taken from [BBB96].

Clients with applications to run contact the local manager to find any idle compute servers. It then connects to the server to run the application. During execution, idle servers steal work from those that are busy.

2.8 Ninflet

Ninflet is a Java based implementation of the Ninf (Network infrastructure for global computing) [TMN98] system which facilitates RPC based computing of numerical task in a wide area network. It was designed to overcome some of the limitations of Ninf providing a more uniform, finer-grained object-based programming model. The Ninflet system consists of three components, a server, dispatcher, and client. The underlying architecture conceptually resembles that of the Javalin system [DoCS96a]. Servers in Ninflet are Hosts in Javalin, the dispatcher in Ninflet is represented as a broker in Javalin, and clients are in both Ninflet and Javalin.

2.9 Popcorn

The POPCORN project provides an infrastructure for coarse grain globally distributed computation over the internet. The infrastructure is designed to harness the millions of processors of the Internet which care to participate at any given moment. A market-based approach of CPU time underlines the system which consist of buyers and sellers of CPU time. CPUs that are idle become the open market, while those that have heavy loads try to sell (or distributed) tasks to other CPUs in the market. The POPCORN programming paradigm achieves parallelism by concurrently spawning off many sub-computations, termed *compuletes.* The underlying system automatically sends these computelets to a market of CPUs. This is maintained by a centralized manager.

2.10 Parallel Java Agents

Parallel Java Agents [KAB98] is a framework for parallel computing in locally confined (i.e. all machines are located in close proximity), scalable computing clusters interconnected by a high-speed LAN. The framework consists of agents which communicate through asynchronous invocations. Agents are dynamically placed (through migration) using an adaptive placement strategy enforced by the underlying software system. The framework has not been implemented, but various computations have been simulated using the adaptive placement strategy. Future development of the system utilizes existing Java Agent systems (e.g ObjectSpace's Voyager [Obj96]).

CHAPTER 3

PROBLEM STATEMENT

Table 3.1 shows a comparison of current Java-based global computing frameworks as discussed in Chapter 2. With the exception of Atlas [BBB96] and Parallel Java Agents [KAB98], none of the previous proposed systems support process migration. Both Atlas and Parallel Java Agents support migration through an underlying software system based on complex scheduling algorithms; which incorporate work stealing and adaptive placement strategies. Each framework is based on an underlying system to distribute processes among processors in the system. ParaWeb [BSST96] allows the programmer to explicitly map processes to processors utilizing a Java Parallel Class Library (*JPCL*). With the exception of IceT [oMCS97], Javalin [DoCS96a], ParaWeb, Ninflet [TMN98], and Popcorn [CLNR97], all frameworks are restricted to execute on distributed systems consisting of High Performance or Scalable Computing Clusters (*HPC/SCC*), Network of Workstations (*NOWs*), or even specific Distributed Shared Memory (*DSM*) architectures (i.e. TreadMarks system for Java/DSM [DoCS97b]). All other systems support global architectures interconnected via the intranet/internet. Most frameworks support a wide variation of programming paradigms. Those based on active objects [BFD96] [SH97] [CM96] [fDRC94] use a deploy on demand method similar to Java Applets [Ham96][DD96] to distribute a computation. Or, they consist of an API which utilize variants of PVM [Sun90], MPI [GLS94], and active objects. Both ParaWeb and Java/DSM use a paradigm that allows processes to share objects through a global name space. Both frameworks require a complex underlying software system to maintain memory coherence among shared objects. Webflow [aSU97b, FF96b] and ARCADE [DoCS97a] programming is based on objects and the dataflow model. Webflow also provides a visual programming tool to develop distributed applications. ParaWeb and ATLAS [BBB96] are based on Java Threads [OW97] and require a modified Java interpreter. Parallel Java Agents [KAB98] is developed on top of ObjectSpace's Voyager [Obj96] Mobile Agent System for scalable computing clusters. It uses a variant of actor [Agh86] communication among agents. With the exception of Parallel Java Agents, and Java/DSM, each

framework uses an underlying system that includes a single scheduling server to remotely distribute threads among distributed computation servers in a network. Although this simplifies the implementation, this strategy is not efficient and fault-tolerant, as the centralized server may become a bottleneck and is a single point of failure. In the following section we present a framework for actor-based distributed mobile computation. This parallel programming paradigm allows computations to be expressed as a set of actors with additional features of mobility and navigational autonomy. This model supports fine, medium, and large grain parallel computations in a scalable high speed interconnection network. It provides an underlying framework for medium/large grain heterogeneous massively distributed parallel computing via the internet.

Table 3.1. Comparison of Global Computing FrameWorks

Framwork	Granularity	Migration	Process Placement	Programming Paradigm	Hardware Architecture	Software Architecture
IceT	coarse	N/A	system	(Active object) using PVM	global	centralized
JavaDC	fine/large (SIMD)	N/A	system	(Active Objects) using PVM/MPI	HPC	centralized
ARCADE	fine/large (SIMD)	N/A	system	dataflow	HPC	centralized
Java/DSM	coarse/large	N/A	system	Shared objects DSM	TreadMarks Machine	complexed memory coherence soft.
Webflow	coarse/large	N/A	system	(Web based) dataflow	mesh of Java servers	centralized
Javlin	large	N/A	system	Java Applets	global	centralized
ATLAS	coarse/large	system	system	Java Threads	NOW	centralized (modified Java interpreter)
Ninflet	fine/large	N/A	system	Java Threads	global	centralized
Popcorn	coarse/large	N/A	system	Active Objects	global	centralized
Parallel Java Agents	fine/large	system	system	actor model	SCC	Voyager Mobile Agent System
ParaWeb	coarse/large	N/A	user/system	Java threads (DSM)	global global	centralized (modified Java interpreter)

3.1 Theoretical Foundation

Multicomputers represent the most promising developments in computer architecture due to their economic cost and scalability. With the creation of many digital high-speed integrated networks [Tan96], heterogeneous multicomputer systems will be used to solve many complex parallel and distributed computations [Hay88, Sta84]. Initially, researchers followed strictly message passing as a common parallel programming paradigm for multicomputer systems. Programmers must explicitly locate processes and communication can be synchronous or asynchronous. Today, several variants

of the message passing paradigm have been proposed within the literature [Agh89, BFD96, BN84, Rie94, Sta84, BN96]. The remote procedure call paradigm provides a method of allowing programs to call procedures located on other machines. Information is transported from the caller to the callee in parameters that are transparent to the programmer. Procedures are statically placed throughout the system to be invoked by remote synchronous/asynchronous procedure calls. The communicating object paradigm regards a distributed system as a set of statically compiled processes communicating with each other via messages. The system's intelligence is embodied in the processes, while the messages contain simple, passive pieces of information. Objects are stationary and communicate through asynchronous messages. This paradigm has been extended to support dynamic or active objects which are deployed on demand through the internet/intranet. Active objects make it easy to write groupware, multiplayer games, and net-centric applications. DSM provides the illusion of shared memory across a loosely coupled system. Processes communicate through the globally shared address space. Each machine which contains a globally shared object must maintain a level of consistency with each replica in the system. Consistency/memory coherence protocols must be used which require additional communication overhead. Recently, autonomous mobile agents have begun to be recognized as a new computing paradigm for distributed systems [Ven97],[Doc95], [KZ97]. An autonomous mobile agent is a program that can change its behavior and migrate from machine to machine in a heterogeneous network dynamically. Mobile agents have the ability to adapt to the computing environment. In this research, we present a communication paradigm among mobile agents that incorporates actor-based message passing to support dynamic architecture topologies for distributed parallel computations. Through mobility agents are able to migrate and adapt to the computing environment. This is ideal for global computation in which applications can take advantage of under utilized resources and locality of reference of data in such systems where the communication bandwidth may be smaller, reliability is lower, and latency is higher. The concept of *actors* was originally proposed by Hewitt [Hew77], and later by Clinger [Cli81], Agha [AKP90, AHP91, AMST91, CA91], Sami [SVN91], Baude [BVN91], Athas [AS88] and in [MC96]. The model has been proposed as a basis for multiparadigm programming in [Agh89] and has been

used as a programming model for multicomputers in [Agh86, BVN91]. Actors are stationary objects that communicate through asynchronous messages. Actors can be dynamically created within a system. We formally define an abstraction of the actor model with the semantics of mobility and navigational autonomy. We express mobile agents as actors using mobile actor semantics. Lastly, we show that our model provides a new solution to programming massively distributed parallel computations. We provide simple examples to illustrate this method.

3.2 The Actor Model

Actors are self-contained, interactive, autonomous components of a computing system that communicate by asynchronous message passing. Actors are characterized by an identity (i.e. mail address), a mailbox, and a current behavior. Moreover, a mail address may be included in messages sent to other actors - this allows those actors to communicate with the actor whose mail address they have received. The ability to communicate mail addresses of actors implies that the interconnection network topology of actors is dynamic. The assumption provides generality: a static topology is a degenerate case in which system reconfiguration is not allowed. This dynamic interconnection network topology implies that the underlying resources can be represented as actors to build a system architecture. Each time an actor processes a communication, it also computes its behavior in response to the next communication it may process. In general, the replacement behavior of an actor may represent the creation of new actors, or a simple change of state variables, such as change in the balance of an account. It may also represent changes in the operations, which are carried out in response to the message. For example, suppose a database actor receives a query request communication. In response, it will perform the query, which will be used to process the next message. A set of procedures and a list of acquaintances define the behavior of an actor. Acquaintances represent actors whose mail addresses are known to the actor. Because all actor communication is asynchronous, all messages are buffered in mail queues until the actor is ready to respond to them. Messages sent are guaranteed to be received with an unbounded but finite delay.

Each actor may be thought of as having two aspects that characterize their behavior:

1. its **acquaintances** which is the finite collection of actors that it directly knows about;

   Diagrammatically we will represent a situation in which an actor W knows about an actor X by drawing a directed arc from W to X (Figure 3.1). W and X are mutual acquaintances [Hew77].

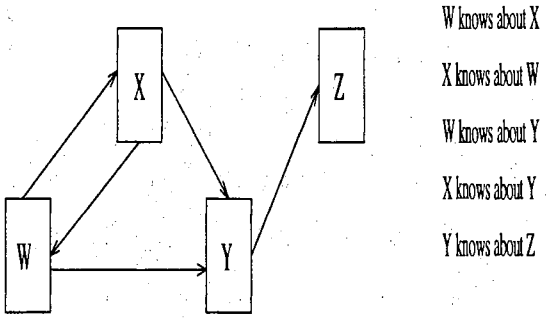


Figure 3.1. Diagram of the acquaintances of actors W,X,Y, and Z.

2. the **action** it should take when it is sent a message. These actions provide a **primitive** set of operations to:

   - *send* messages asynchronsly to specified actors,
   - *create* actors with specified behaviors, and
   - *become* a new actor, assuming a new behavior to respond to the next message.

All computation in an actor system is the result of processing messages. Therefore, an actor's behavior is a function of the message accepted. Figure 3.2 illustrates the behavior of an actor in response to a message[Agh86]. The **become** primitive is used as a mechanism for changing the current behavior, and for indicating when the actor may begin processing the next message. [AKP90] proposed a **call/return** communication operator which provides a simple abstraction to express process dependence using synchronous communication. In call/return communication, an object invokes another object and waits for it to return a value before continuing. This is

analogous to a synchronous remote procedure call (RPC) [AS88]. In order to achieve maximal concurrency, we generally do not want to block a sender of a call/return message send; if the actor invoked is located on another node. Therefore, it has been shown in [AKP90] that call/return communication can be transformed to semantically equivalent asynchronous message sends with corresponding continuations.

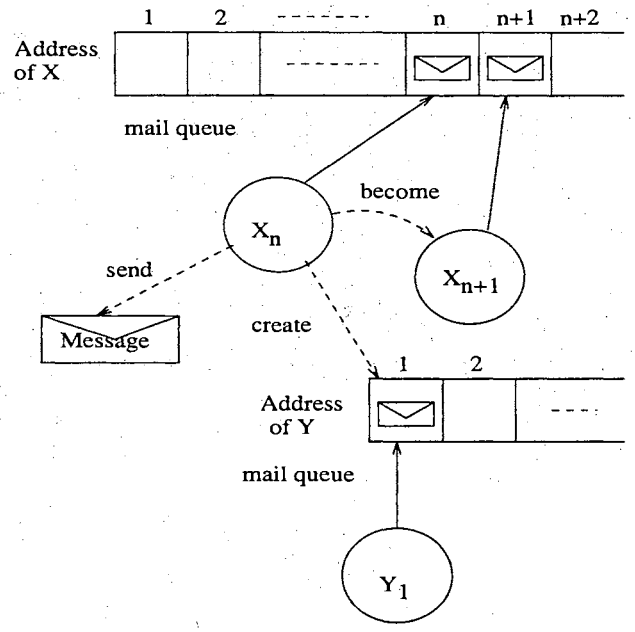


Figure 3.2. Actions performed by an actor in response to a communication.

3.3 Motivation

The actor primitive operators (i.e. *send, create, and become*) form a simple but powerful set on which to build a wide range of higher-level abstractions and concurrent programming paradigms. In this paper, we present a communication paradigm among mobile agents that incorporates actor-based message passing to support dynamic architecture topologies for massively distributed parallel computations via the Internet. We build upon the actor primitive operators and extend the semantics of the actor model to support actor mobility and navigational autonomy. Currently, message passing models provide process mobility/migration through transportable and autonomous mobile agents

[BFD96, Gra95, Inc94, Whi94b]. Although agent technology is commercially in use, there has been no formal characterization of agent performance. Researchers and developers assume that mobile agents consume fewer network resources than the client/server models, and that they are feasible and efficient to support large scale distributed applications [oCaI96]. Although the assumption clearly holds for specific applications [BFD96, Gra95, Inc94, Whi94b], the range of applications for which it holds is unknown. Therefore, it is difficult to choose a communication paradigm for a distributed application [oCaI96].

The behavior of the actor model is formally characterized as a feasible and efficient model to program fine grain parallel applications on multicomputers with a high speed interconnection network. It has been proposed as a concurrent programming language model due to its inherent concurrency in the evaluation of expressions [Agh86, AKP90, Agh89, AHP91, MC96]. It has been shown that support for large-scale concurrent systems requires building on simple programming primitives incorporated into the actor model to form multiparadigms [Agh89]. Actor architectures for solving fine grain applications on multicomputers [Agh86, AHP91, AS88], as well as, actor-based frameworks for solving large scale applications in a heterogeneous computing system [AP91] have also been proposed. Concurrency issues such as: divergence, deadlock, and mutual exclusion have been addressed in [Agh86], along with the mathematical theory of computation that any kind of discrete behavior can be physically realized [Hew77]. Although there is sufficient research supporting the actor model to solve fine/large grain applications on a tightly coupled system, there has been no actor-based solution to solve large scale data intensive distributed applications which may be interconnected by costly communication links. In order to support this environment, locality of reference and resource management (i.e. load balancing) must be addressed; as processes must be able to migrate throughout the system. In the next section, we address the issue of locality of reference and resource management through actor mobility. Current actor-based systems require a tightly coupled underlying software system which implicitly distributes actors among processors in the system, as well as transparently provides global access to all behaviors in the system. Through actor mobility no centralized software system is needed to provide access to all behaviors in the system. Actors and

their behaviors are distributed throughout the system explicitly by the programmer using mobile actor constructs.

3.4 The Mobile Actor Paradigm

In this model, all processes in a heterogeneous system are considered to be actors. There may be several actors mapped to a processor. Actors encapsulate a single process which is the only process that may be used to execute an actor's behaviors. This prohibits multiprocessing within actors. Actors may be partitioned into classes namely *primitive* and *non-primitive*.

- *Primitive* actors correspond to the usual atomic types such as numbers and characters. They are sent directly in messages. Primitive actors are immutable. Their identity may be represented by their state (i.e. the behavior of an actor is the same always and everywhere) [Agh86].

- *Non-primitive* actors have an identity that is represented by a reference, a current behavior that includes the methods that define the actions that the actor can take upon receipt of a message, and a set of acquaintances that the actor can communicate with. When a non-primitive actor is sent a message, it is actually the reference to a behavior and it's arguments, if any, that is sent (Figure 3.3). The behavior being a procedure (method) stored on the local machine of the recipient. Since all actor communication is asynchronous, the method of communication is analogous to the asynchronous remote procedure call (RPC) [AS88].

Figure 3.3 illustrates a communication send from $Actor_X$ to $Actor_Y$. In response to the communication, $Actor_Y$ assumes the behavior $f()$; which executes and returns the result. Notice that the behavior $f()$ must initially reside on the remote machine B. Step *(1)*: $Actor_{X_n}$ on machine A sends the communication $[f(), args..]$ to $Actor_Y$ on machine B; $Actor_{X_n}$ then becomes a new actor $Actor_{X_{n+1}}$ (possibly itself or to await the result). Steps *(2)* and *(3)*: $Actor_{Y_m}$ receives the communication $[f(), args..]$, and in response to the communication assumes the behavior $f()$. Step

(4): $Actor_{Y_m}$ returns the result to $Actor_X$ on machine A. Lastly, step *(5)*: $Actor_{Y_m}$ becomes $Actor_{Y_{m+1}}$ (possibly itself).

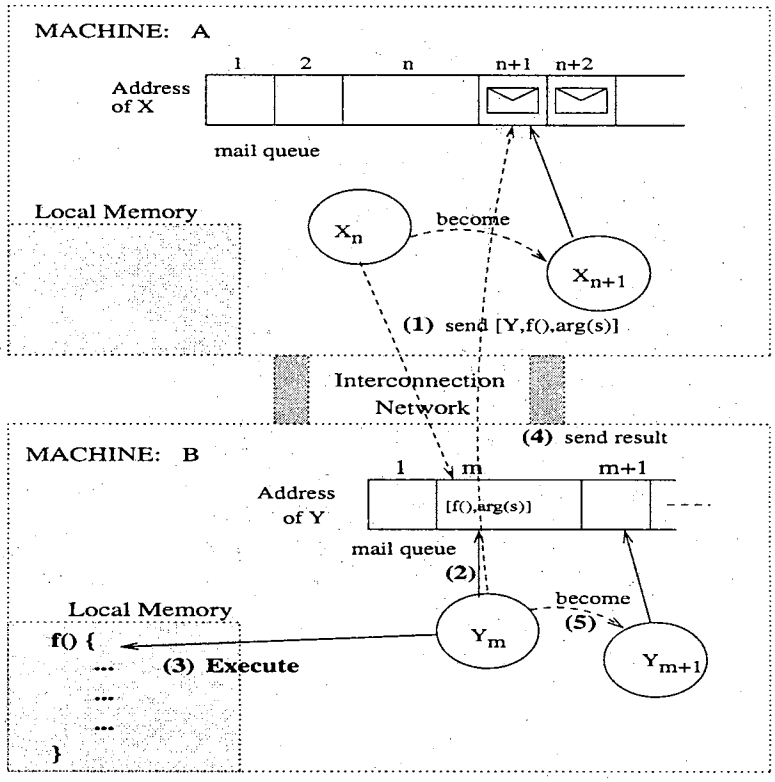


Figure 3.3. $Actor_X$ **sends** $Actor_Y$ **a message referring to a behavior** $f()$**.**

High-level communication, synchronization and coordination abstractions can be formed from primitive/non-primitive actors collectively. Some examples are synchronization *barrier actors* or *mutex actors* [AP91]. The *barrier actor* allows processes to synchronize at a specified point within an execution. A *mutex actor* would provide synchronization access to a critical section. By developing these hich level abstractions, it is easy to address concurrency issues such as: divergence, starvation, deadlock and mutual exclusion [Agh86].

A *mobile actor* is a non-primitive actor with the semantics of mobility and navigational autonomy. Navigational autonomy is the degree to which a message can be viewed as an object with its own innate behavior, capable of making decisions about its own destiny. The actor model inherently enforces navigational autonomy allowing addresses of actors to be communicated and thus providing a dynamic interconnection network topology. Such a computing model provides support

to deal with non-deterministic problems which require network reconfigurations, non-deterministic communication, and dynamic process coordination.

An important degree of flexibility available in actor semantics involves the ability to carefully control the articulation of details to be included in specifications. That is, the constraints on the behavior of a system of actors can be specified in as much or as little detail. Therefore, the behavior of an actor can be characterized as large, medium, or fine grain. In many practical distributed applications, the over consumption of local resources don't allow computations to be processed efficiently. A more feasible solution would be to migrate the process to least consumed resources, or to move the process to a data server or communication partner in order to reduce network load by accessing a data server or communication partner by local communication. We propose a strategy for remote execution and process migration using the actor-message passing paradigm (i.e. for load balancing, and locality of reference of data/behaviors). A remote execution includes the transport and start of execution of a process on a remote location. Process migration includes the transport of process code, execution state, and data of the process; processes may be restarted from their previous state.

We extend the actor primitive operations in response to a message with semantics to support actor mobility. The semantics of actor mobility are enforced: upon receipt of a message, or when dynamically creating another actor on a remote location. These extended primitive operations allow computations to migrate after explicit checkpointing by the programmer, or the underlying system. The behavior of mobile actors consists of two kinds of actions in response to a message:

1. $become_{remote}$ computes a replacement behavior on the local machine and migrates to a location on a remote machine. The migrated actor is characterized by the identity (i.e. it's mail address), and mailbox of a specified location of an actor on a remote machine.

2. $create_{remote}$ a new actor on the local machine and migrate to the remote location, assuming a new behavior to respond to the next message.

Actors and their behaviors to be migrated are known as *carried-functions* and can be in the form of source code, or native bytecode [Ham96, Inc95]. Information sent along with the carried-function are its arguments along with the process state information. The $become_{remote}$ primitive operator causes an actor to receive the local communication, bind to a behavior in response to the communication, migrate the actor along with it's behavior to the new location, then continue processing at the new location. The new location is the local machine of an acquaintance. The migrated actor assumes the identity and mailbox of an actor which resides at the new location on the remote machine. The migrated actor no longer exist on the origination machine. This allows the migrated actor to take advantage of accessing local resources (i.e. data, and communication with other local actors) at remote locations.

The $become_{remote}$ primitive operation provides a convenient method for actors to migrate when replacing the current behavior in response to a communication. Explicit checkpointing (i.e. for load balancing and locality of reference) within the code, or from the underlying system could occur before a $become_{remote}$ operation; allowing the actor to migrate to the best possible location. Consider the case where an actor needs to process data maintained on a remote machine. Actors which reside on the remote machine do not maintain the behavior to continue processing the data. Data can either be sent to an actor with the desired behavior located on another machine (i.e. using the *send* primitive). Or, an actor who resides on the same machine as the needed behavior could be sent a message to reference the remote data. In response to the communication, the replacement behavior could be migrated to the remote machine for processing. The semantics of the $become_{remote}$ operation are the same as the original *become* primitive, if the remote identity and machine are the same as the initiating actor's identity.

Figure 3.4 illustrates a $become_{remote}$ operation from $Actor_{Y_{m-1}}$ on machine B. $Actor_{Y_{m-1}}$ becomes a new $Actor_{Y_m}$ and assumes a new behavior $f()$. $Actor_{Y_m}$ migrates to machine A assuming the identity/mailbox of X. All replacement behaviors are computed in response to communications sent to $Actor_X$ on machine A. Notice $f()$ is migrated to machine A and does not need to initially reside on machine A. Step *(1)*: $Actor_{Y_{m-1}}$ which assumes behavior $g()$, executes a $become_{X:A}f()$

operation. Step *(2)*: $Actor_{Y_{m-1}}$ becomes a new $Actor_{Y_m}$, and assumes a new behavior $f()$. Steps *(3)* and *(4)*: In response to communication m, $Actor_{Y_m}$ migrates to machine A assuming the identity/mailbox of $Actor_X$. Step *(5)*: $Actor_{X_n}$ continues executing its current behavior $f()$. Lastly, Step *(6)*: $Actor_{X_n}$ computes a replacement behavior in response to communication $n+1$.

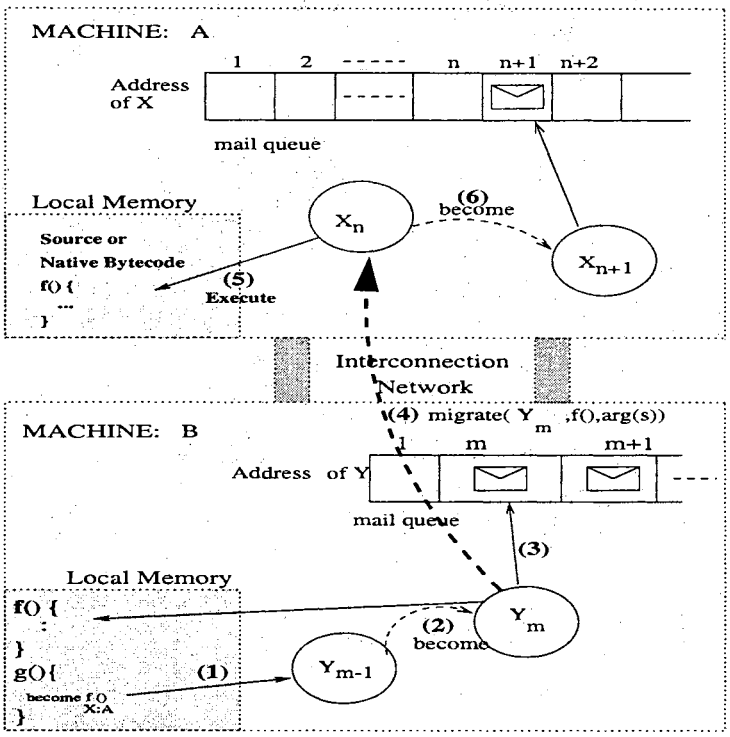


Figure 3.4. $Actor_{Y_{m-1}}$ **executes a** $become_{remote}$ **operation.**

The creation of actors is not limited to the domain of the local machine, but may span the global system. The $create_{remote}$ primitive operation causes the migration of a dynamically created actor with specified behavior(s) (i.e. its mail queue along with its behavior is created and transferred to the remote location). It creates a unique mail address which is returned to the initiating actor. Through the use of the $create_{remote}$ operation, programmers are also able to explicitly map actors to machines. This is useful when statically placing actors throughout the system. Actors are also able to spread themselves dynamically across a global system. This may occur if behavior methods contain $create_{remote}$ statements; allowing actors created remotely to create other actors on remote machines, and so on. It is also useful during load balancing in which actors must migrate to least

consumed processors. This could occur after an explicit checkpoint within the code or from the underlying system.

Figure 3.5 illustrates a $create_{remote}$ operation executed as a statement in behavior $g()$ by $Actor_{X_n}$ on machine A. The remote location is machine B. The new actor's identity is Y on machine B, and it assumes the new behavior $f()$ in response to a communication. Notice $f()$ is migrated to machine B and does not need to initially reside on machine B. Step *(1)*: $Actor_{X_n}$ executes the statement $create_B f()$; to create a remote actor assuming the new behavior $f()$ on machine B. Step *(2)*: The new $Actor_Y$ along with its mailqueue is created locally on machine A. Step *(3)*: $Actor_{Y_1}$ along with its mailqueue are migrated to the remote machine B. Steps *(4)* and *(5)*: $Actor_{Y_1}$ receives a communication, and assumes the behavior $f()$. Lastly, Step *(6)*: $Actor_{Y_1}$ computes its replacement behavior (possibly itself) in response to communication 2. When developing a mobile actor system, newly created remote actors could construct mailqueues after migration occurs. This is an implementation issue.

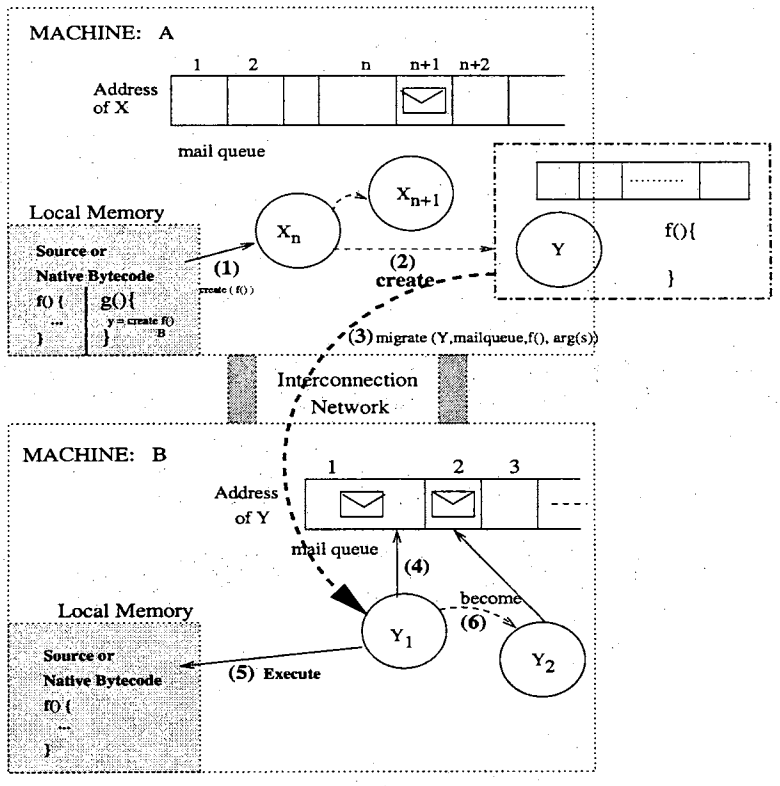


Figure 3.5. $Actor_X$ **creates a remote** $Actor_Y$**.**

CHAPTER 4

JMAS: A JAVA-BASED MOBILE ACTOR SYSTEM

4.1 Properties of Global Systems

Exploiting the resources of millions of computers on the Internet to form a powerful global computing infrastructure is the goal of this research. Such an infrastructure should provide a single interface to users that provides large amounts of computing power, while hiding from users the fact that the system is composed of hundreds to thousands of machines scattered across the country. Our vision is to create a system in which a user sits at a workstation, and has the illusion of a single very powerful computer. In this section, we discuss the technical issues associated with the construction of a global computing infrastructure which executes mobile actor computations. A mobile actor system is a multi-user, heterogeneous, global computing environment for executing distributed actor-based computations. A mobile actor system must support two basic tasks - the creation and migration of remote actors, and the communication between actors distributed throughout the system. In addition the system should:

- provide language support for the mobile actor programming model,
- provide a single consistent namespace for actors within the system,
- provide an efficient execution schedule between actors maintained on the local machine,
- be able to distribute the load evenly among the machines participating within the distributed system,
- exploit heterogeneity,
- be fault tolerant, and
- be secure.

4.1.1 Language Support

The programming language used in a mobile actor system should support the underlying computation model on heterogeneous machines; therefore, requiring the language to be interpreted to provide portability. The language should be easy to use; allowing the user to link in the desired libraries, or use constructs provided by an Application Programming Interface (*API*). Language support should provide constructs to create remote actors throughout the distributed system, as well as, send communications between actors within the system. Communication in mobile actor systems is point-to-point, non-blocking, asynchronous, buffered, and may consist of complex objects. The language should be able to support the mobile actor communication paradigm. The language should support static or dynamic placement of actors throughout the distributed system.

4.1.2 Exploiting Heterogeneity

Creation of remote actors requires the migration of code along with its state information. In order to support heterogeneity, migrated code needs to be in a format that is not dependent on the underlying system. Therefore, requiring the need and use of native machine code or bytecode [BFD96]. Each machine within the system should contain a server that sends and receives migrated code and state information of mobile actors.

4.1.3 Consistent Namespace

Sending a message requires the receiver's current locality be known to the sender. In a mobile actor system, an actor's where-abouts are abstractly represented by its mail address. The entities used to define a mail address determine the efficiency of name translation, as well as, the degree of location transparency. There are two strategies that could be used:

1. Use of location-dependent entities tightly coupled with actors offers efficient name translation at the expense of location transparency.

2. Location-independent entities allow location transparency but increase name translation time.

4.1.4 Scheduling and Load Balancing

Scheduling and load balancing policies must accommodate the heterogeneous and distributed nature of the mobile actor system. An efficient method for automatically scheduling parallel subcomputations across the distributed system should insure that all machines within the distributed system are fully utilized; taking advantage of idle CPUs. In general, such a method will need to be adaptive and may require keeping track of the load on different machines and the communication patterns between different actors. Scheduling on the local system, must insure no starvation among local processes.

4.1.5 Fault Tolerance

In a global system, it is certain that at any given instant several machines, communication links, and disks will have failed. Thus dealing with failure and dynamic reconfiguration is a necessity. There is a trade-off between performance and different levels of fault tolerance. Fault tolerance can occur within the system itself, or within the application. Fault tolerance should be addressed to the extent necessary without compromising the performance of the system. In most systems, it is desirable to just consider fail-stop faults of hardware components, including both processors and the network.

4.1.6 Security

The issue of security is of foremost concern. In most global computing systems, participation requires either downloading and running an executable, or downloading unknown source code and then compiling it. The code may contain bugs or viruses that can destroy or spy on local data. In addition, objects that are migrated between machines within the system may also contain bugs or viruses. Because mobile actors can migrate from machine to machine, security policies should be enforced to insure the actor is valid. Such as:

- using cryptographic authentication protocols, or
- using digital signatures.

In addition, all messages communicated within the system should be encrypted. Providing security policies compromises the performance of the system; because, all communications and migrated code must be encrypted before transmitted, and decrypted before being processed.

4.2 JMAS Infrastructure

JMAS is a globally distributed computing environment for executing mobile actor computations. JMAS is designed using Java technology [Inc95], and requires a programming style different from commonly used approaches to distributed computing. JMAS allows a programmer to create mobile actors, initialize their behaviors, and send them messages using constructs provided by the JMAS Mobile Actor API. As the computation unfolds, mobile actors have the ability to implicitly navigate autonomously throughout the underlying network. New messages are generated, new actors are created, and existing actors undergo state change. JMAS also makes mobile actor locality visible to programmers to give them explicit control over actor placement. However, programmers still do not need to keep track of the location to send a message to a mobile actor. Data flow and control flow of a program in JMAS is concurrent and implicit. A programmer thinks in terms of what an actor does, not about how to thread the execution of different actors. Communication of mobile actors is point-to-point, non-blocking, asynchronous, and thus buffered.

4.2.1 Language Support in JMAS

JMAS is based on the Java Programming Language and Virtual Machine of JDK1.1 [Inc95]. JDK1.1 contains mechanisms that allow objects to be read/written to streams (object serialization)[atoSM96a], as well as, an API that provides constructs to dynamically build objects at run-time (i.e. Reflection package [*java.lang.reflect*]). We exploit heterogeneity through Java's platform independent (i.e. write once run anywhere) framework. We provide a Mobile Actor API for developing mobile actor applications using the Java Programming Language. Mobile actor programs are compiled using a Java compiler that generates Java bytecode. Java bytecode can be executed on any machine containing a Java Virtual Machine. Actors in JMAS are light-weight processes called threads. The API provides constructs which allow programmers to create mobile actors using static or dynamic

placement, to change an actor's state, and to send an actor communications.

4.2.2 Consistent Mobile Actor Names in JMAS

JMAS implements a simple location-dependent naming strategy tightly coupled with mobile actors within the system. Each mobile actor within the system is given a globally unique identifier. This identifier is bound to only one address by the underlying message system. These bindings may change over time; if for example, a mobile actor migrates to a different machine. In such a case, messages are forwarded to the new location by the underlying message system. It has been shown in [BN95], that forwarding messages in a distributed system consisting of N machines requires in the worst case $N-1$ message rounds. Our strategy performs fast creation and translation of globally unique identifiers using the naming function below:

$$N(ObjectName) = ObjectName + Time + orig + counter@dest$$

Where *Time* is the current time on the local system, *orig* is the location which invoked the creation of the new actor, *counter* is a monotonically increasing local counter, and *dest* is the destination machine where the new actor will be created. As illustrated in Figure 4.1, actors which are created within the system must first register with the local nameservice. Each actor whether created local or remote maintains a globally unique name. Although this is a simple approach, the disadvantage is that identifier names could become long, increasing the communication overhead.

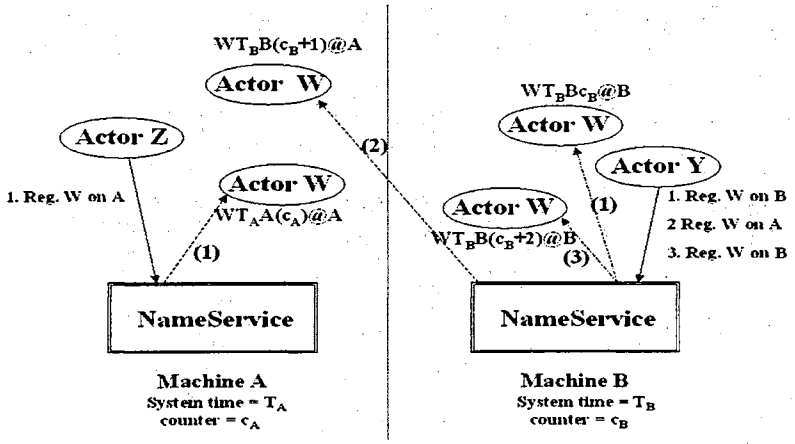


Figure 4.1. Creating Globally Unique Actor Names.

4.2.3 Scheduling and Load Balancing in JMAS

The JVM implements a timeslice schedule of threads on Window95 systems, and a pre-emptive priority-based schedule for UNIX/Windows NT systems. JMAS forces a pre-emptive, priority-based schedule among threads; regardless of the underlying architecture. The efficiency of an actor-based computation on a loosely coupled architecture depends on where different actors are placed and the communication traffic between them. Thus, the placement and migration of actors can drastically affect the overall performance. We implement a decentralized fault-tolerant load balancing scheme based on the CPU market strategy proposed in [CLNR97]. The market strategy is based on CPU-time. Entities within the system consist of *buyers* and *sellers*. A *seller* allows its CPU to be used by other programs. A *buyer* serves as a machine wanting to off-load work to a seller. A meeting place in which buyers and sellers are correlated is known as a *market*. This strategy is intended for coarse-grain applications.

4.2.4 Security in JMAS

Security issues are not addressed in this prototype system. Policies could be enforced to encrypt/decrypt all Java class files and messages sent throughout the system. Use of any strategy will compromise the overall performance of the system.

4.2.5 Fault Tolerance in JMAS

Machines used within the JMAS infrastructure are fault tolerant to the extent necessary without compromising overall system performance. The limit of our concern is with fail-stop faults of hardware components, and the network. The underlying communication system will guarantee the delivery of messages through the use of reliable, communication-oriented TCP sockets[Tan96]. Further, if a host should fail, then JMAS will remove that host from the current CPU Market configuration. Software faults are handled through the use of Java Exceptions [Inc95] and are not of our concern.

CHAPTER 5

JMAS ARCHITECTURE

The architecture of JMAS is organized as a series of layers or levels, each one built upon its predecessor (Figure 5.1). The lowest level(**physical layer**) is the actual physical network, which may consist of a LAN/WAN of PCs and/or workstations. It could also represent a global network such as the Internet. The second layer (**daemon layer**) consists of the collection of daemons residing on all physical machines participating in the distributed system. Each daemon listens on a reserved communication port receiving communications that could consist of messages or migrating computations. Upon receipt of a communication, it is passed to the third layer. The third layer consists of Distributed Run-Time Managers (**D-RTM**). The D-RTM is responsible for message handling from/to local/remote processes, scheduling and load balancing of processes. The forth layer (**logical layer**), consist of the actual application specific computations on the local machine. Computations are expressed as *mobile actors*. Each actor is encapsulated with a behavior, an identity, a mail queue, and one thread. The logical layer shows each actor and its acquaintances (i.e. A knows about B and C, ...etc).

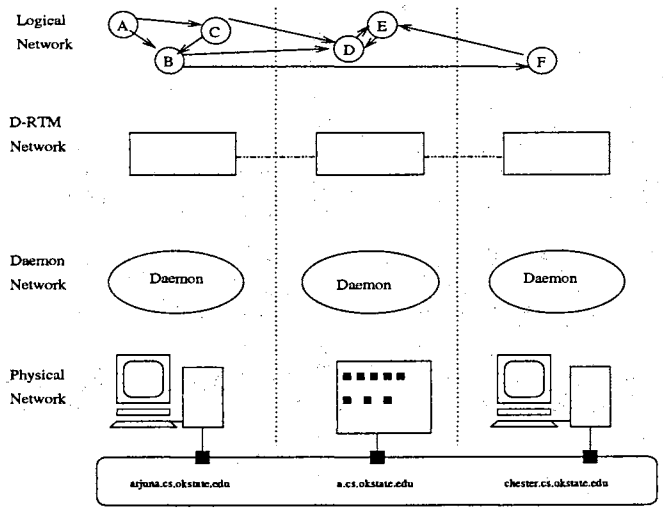


Figure 5.1. Four Layer Mobile Actor Architecture.

In the following sections, we give a detailed description of the JMAS architecture. In particular, we discuss the components of each layer, and show how Java technology is applied.

5.1 Physical Layer

The physical layer is the actual physical network, which may consist of a LAN/WAN of PCs and/or workstations. These systems are referred to as scalable computer clusters (SCCs), or networks of workstations (NOWs) [ACP95]. Both systems are developed within a trusted environment. Therefore security issues are not a major concern. The disadvantage is that the scalability of these systems is limited to the resources available to the system administrator. The physical layer could also represent a global network such as the Internet. A global framework is dynamic, and scales to millions of machines. This creates an unsecured environment; prone to malicious mobile code, and computer hackers. Security is a major concern. Architects of global systems provide security using encryption/decryption techniques on all communication messages and mobile code within the system. The JMAS prototype was developed for testing in a secure, trusted environment. As shown in Figure 5.2, users of a global system can have different logical views of the underlying physical network; and views may overlap.

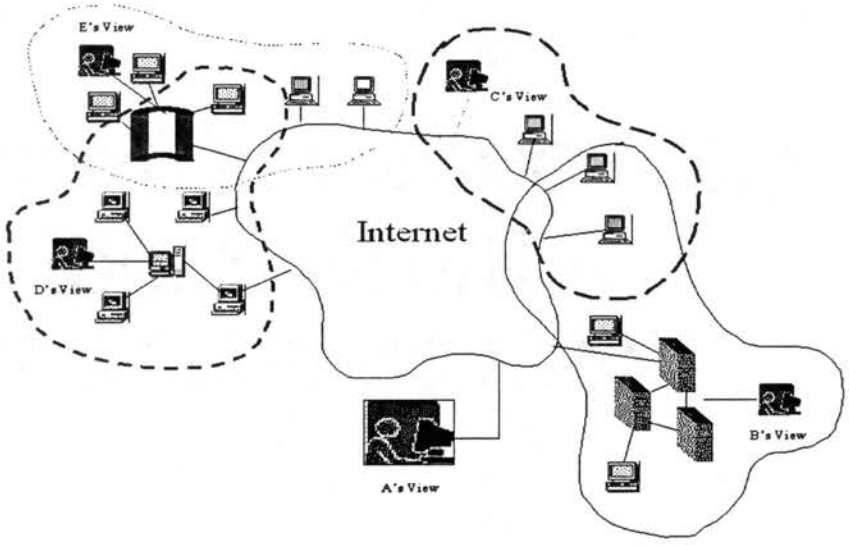


Figure 5.2. Users Logical View of Global System.

5.2 Daemon Layer

The daemon layer is implemented as a collection of daemon threads residing on all physical nodes participating in the JMAS distributed environment. The responsibility of the daemon thread is to continuously monitor the network, receiving local/remote communication messages and mobile computations arriving from other machines. JMAS supports a messages-driven model of execution (Figure 5.3). There is no local/remote peer-to-peer communication between mobile actors within the system. All communication is routed through a reserved port of a daemon thread residing on the local machine. The reserved port for JMAS is *9000*. Message reception by the daemon thread creates a thread within the actor which executes the specified method with the message as its argument. Only message reception can initiate thread execution. Furthermore, thread execution is atomic. Once successfully launched, a thread executes to completion without blocking.

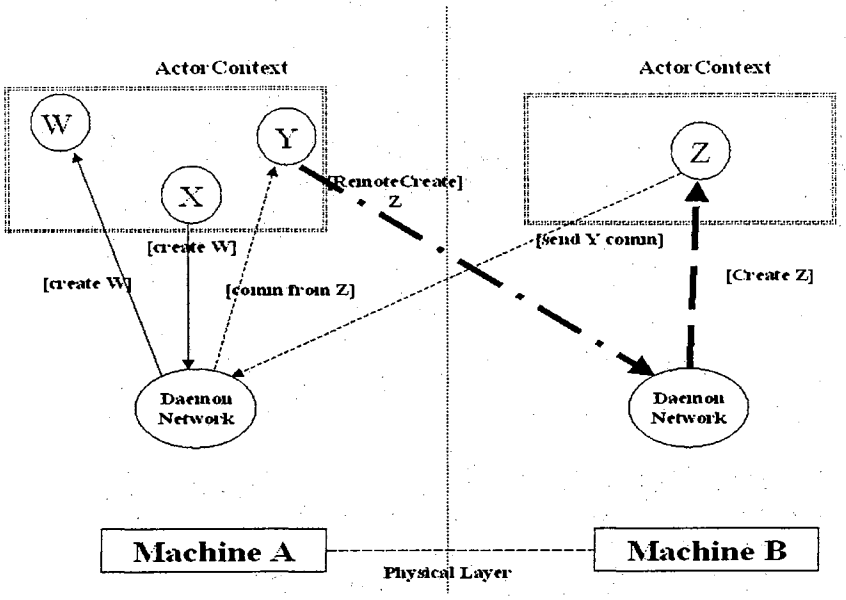


Figure 5.3. Message-driven model of execution.

Communication messages in JMAS are denoted as **Tasks**. As shown below, a Task contains information such as:

- the destination of the Task.
- the origination of the Task.
- the type of Task:
  - a communication from another mobile actor, or
  - remote Java bytecode and/or a migrated process.

5.3 Distributed Run-Time Manager

The Distributed Run-Time Manager (D-RTM) is the most complex of the four layers. It is contained within each daemon in the system. Therefore, the daemon layer and D-RTM layer are tightly coupled. The D-RTM contains the basic underlying software that provides the transparent interface to the global system. The D-RTM was designed using a layered virtual machine design built on top of the Java Virtual Machine (JVM) using JDK1.1 [Inc95] (Figure 5.4). The basic sequence of operations carried out by the D-RTM is shown in Figure 5.5. The D-RTM has several functions:

- To handle all incoming Tasks (i.e. **Message Handler**)
- To prepare actor processes to run on the local system (i.e. **Actor Context**)
- To load java bytecode (e.g. java objects) from local/remote locations(i.e. **BehvLoader**)
- To schedule local/remote threads using a pre-emptive, priority schedule (i.e. **Scheduler**),
- To manage the CPU load on the local machine (i.e. **Load Balancer**).

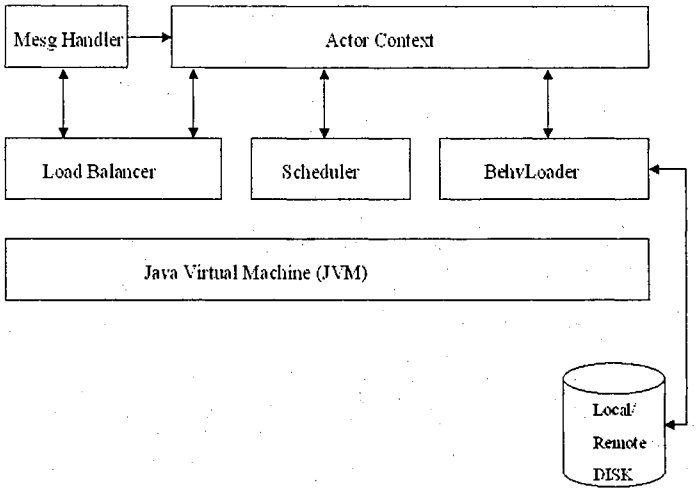
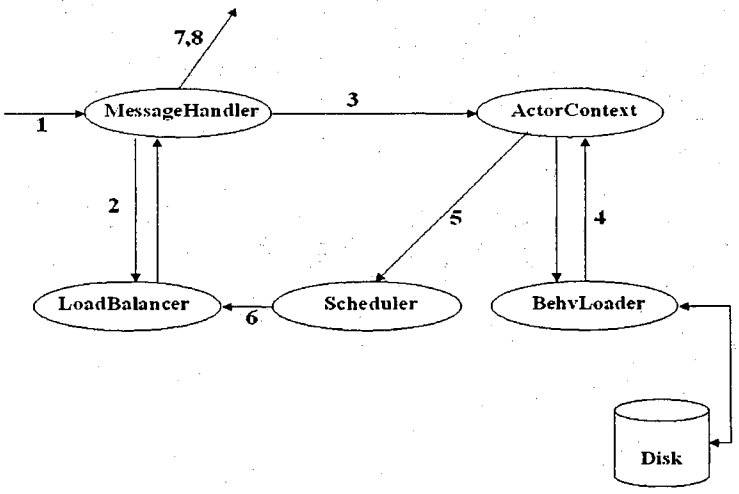


Figure 5.4. Distributed Run-Time Manager (D-RTM).



1. Receive Task
2. If Task is an actor
   - Check load
   - If load > Threshold, then goto 7

   Else if Task is a communication
   - Check Message Handler queue for actor; If forwarded, goto 8
3. If actor, block until communication received, OR If communication, block until actor received
   - Prepare actor to run
4. Load class into interpreter
5. Schedule actor
6. Inform LoadBalancer, goto 1
7. Forward Task, update Message Handler queue of new location, goto 1
8. Forward Task to new location, goto 1

Figure 5.5. Process Flow Diagram of D-RTM.

5.3.1 Message Handler

The message handler is responsible for routing Tasks which consist of communications to local actors. As illustrated in Figure 5.6, messages are stored in a table of message queues (i.e. mailboxes). A mailbox could have one or more actors within the local actor context associated to it. We implement the table of mailboxes as a hash table. We use Java's Hashtable class provided by the *java.util* package. Because Java implements its Hashtable as a synchronized object, each access to the Hashtable is atomic. This is very useful for our multi-threaded environment. Each mail address hashes to one mailbox in the table. In order to achieve maximum parallelism, the table is accessed by subprocesses. Messages from a desired mailbox are forwarded asynchronously to actor processes whose identity is denoted by the mail addresses of the mailbox.

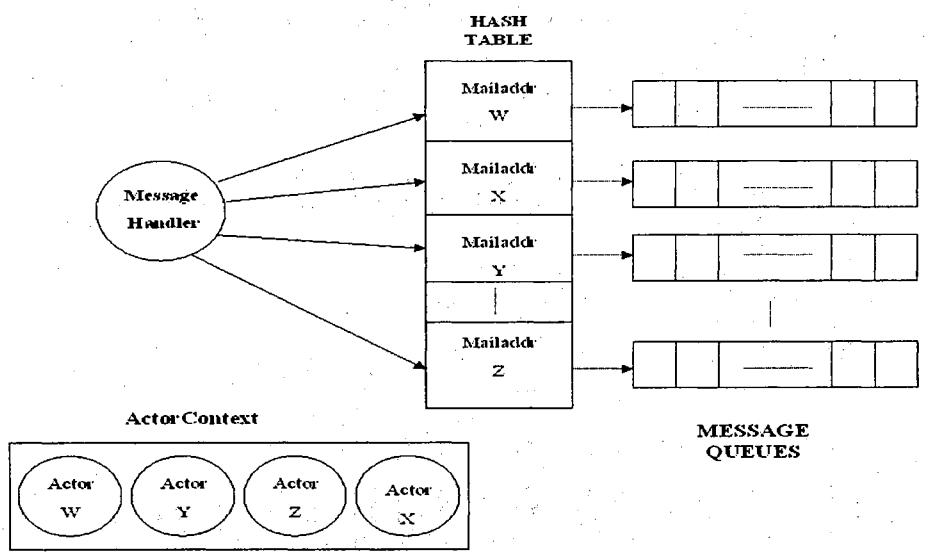


Figure 5.6. Message Handler.

5.3.2 Actor Context

The Actor Context is responsible for instantiating an object, wrapping the object within a thread, and supplying the thread to the Scheduler. It also maintains a table of system information. Such as:

- The actor Identity
- The current behavior
- The current method (communication being executed)
- The total (idle) time actor waited in ready queue before receiving a communication (msec)
- The total time to load the actor (msec)
- The current running time (msec)

Objects in JMAS are built during runtime. Information about an object during runtime is obtained using Java Reflection [Inc95]. The classes needed to perform these operations are obtained from the *java.lang.reflect* package of the JDK1.1. Figure 5.7 illustrates the procedure of instantiating an object *HelloWorld* and calling its *hello* method.

Example:

```
// Instantiating an Object, calling its constructor

Class cl = loadClass("HelloWorld");
Object[] arg = new Object[] { new String("hello") };
Class[] param = new Class[arg.length];
for(int k = 0;k< arg.length;k++)
param[k] = arg[k].getClass();

Constructor x = cl.getConstructor(param);
x.newInstance(arg);

// Calling one of the object methods

Method m;
m = cl.getMethod("hello",null);
m.invoke(null,null);
```

Figure 5.7. Building Objects at Run Time.

5.3.3 Scheduler

JMAS implements a pre-emptive, priority-based scheduler among threads. Each thread is assigned a priority that can only be changed by the programmer. The thread that has the highest priority is the current running thread. Processes with a lower priority are interrupted. To ensure that starvation

does not exists among threads we implement a round-robin schedule among local processes. As illustrated in Figure 5.8(a), incoming threads or threads instantiated locally, are given a priority–initially low. Threads are then placed into a queue data structure. The scheduler dequeues a thread from the list and assigns it the highest possible priority–causing the this thread to run. After a given time t, the thread is stopped and inserted back into the list. This process continues until all threads within the list terminate (Figure 5.8(b)). The scheduler could be interrupted by the load balancer; if the CPU reaches its computation threshold. This will cause the current running thread to suspend and migrate to a remote machine to continue its execution.

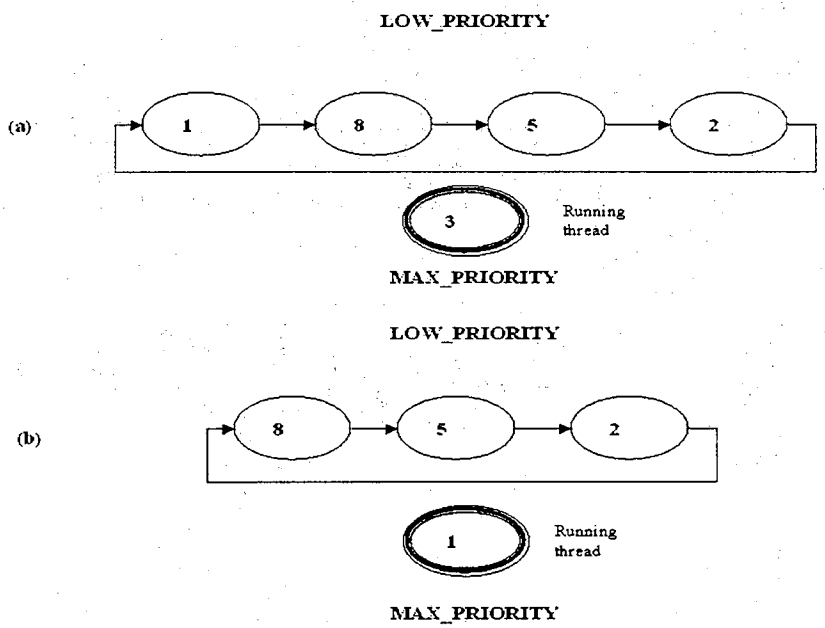


Figure 5.8. Thread Scheduler.

5.3.4 ClassLoader

Objects needed for execution in a program are loaded to the Java interpreter by the Java *classloader*. As shown in Figure 5.9, the Java Classloader loads classes to the interpreter using the following sequence of operations:

1. Check if the class already exists in the local cache. If not,

2. check if the class is a system class. If not,

3. check the local disk. If not found,

4. *NoSuchClassFound* exception is thrown.

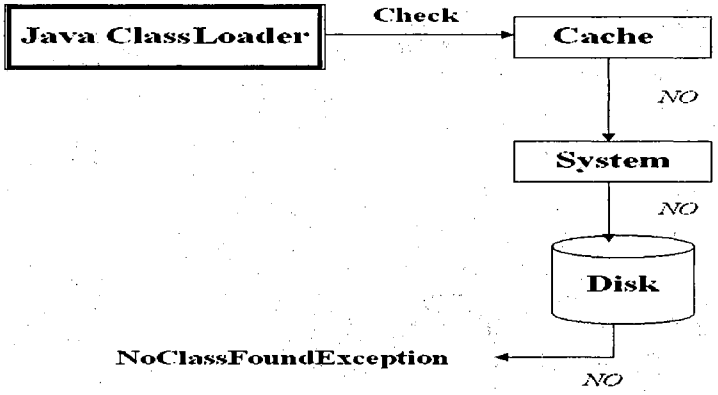


Figure 5.9. Operation of Java ClassLoader.

In order to work around the security restrictions provided by the JVM and to load classes from remote locations, we implemented our own classloader. The *BehvLoader* allows classes to be loaded over the network and stored within the local cache. The BehvLoader loads classes to the interpreter using the following sequence of operations (Figure 5.10).

1. Check if the class already exists in the local cache. If not,

2. check if the class is a system class. If not,

3. Check the local disk. If not found,

4. check the remote disk where the request originated. If not found,

5. *NoSuchClassFound* exception is thrown.

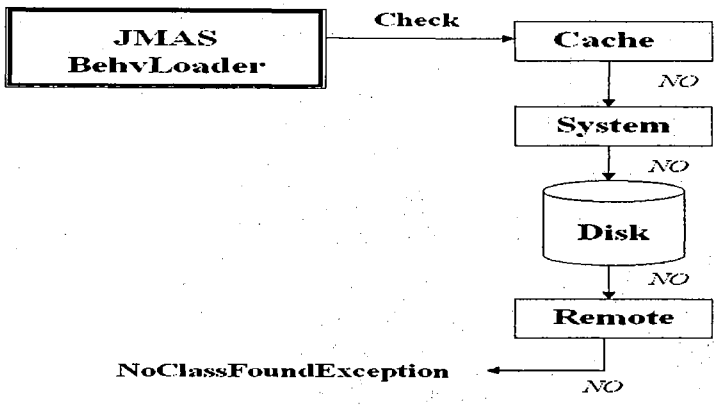


Figure 5.10. Operation of JMAS ClassLoader.

Different features can be added to the BehvLoader to provide security. Such as:

- encryption/decryption of class files
- use of signatures

5.3.5 Load Balancer

We implement a load balancing scheme based on the CPU market strategy proposed in [CLNR97]. The market strategy is based on CPU-time. Entities within the system consist of *buyers* and *sellers*. A *seller* allows its CPU to be used by other programs. A *buyer* serves as a machine wanting to offload work to a seller. A meeting place in which buyers and sellers are correlated is known as a *market*. CPUs are chosen from the market using three selection policies:

1. Optimal (Best) selection,
2. Round-Robin selection, or
3. Random selection.

Developing a Market of CPUs

We implement a decentralized hierarchical method for organizing the CPU market. Each machine within the system is responsible for managing a market. Therefore, the process of managing a

market is distributed throughout the system–increasing market reliability and availability. When starting the system, the D-RTM initializes its market by registering itself with machines designated within a configuration file set by the system administrator. Those machines willing to sell their CPU respond with a message *SELLER*, and are added to the market as *sellers*. Machines who wish to buy CPU time respond with a message *BUYER*, and are added to the market as *buyers*. Those who do not respond (i.e. system down) are not added to the market. This market maintained by the D-RTM, contains the secondary machines on which to off-load remote processes. As shown in Figure 5.11, this creates a logical hierarchy of machines. Each node within the hierarchy, with the exception of the bottom most nodes, are denoted as market managers. Communication overhead is minimal. CPUs wishing to sell their time add themselves to the market by notifying a market manager (Figure 5.11). Buying from the market is a bottom up process. Nodes at the lowest level become overloaded faster. Once a given node X is denoted as a buyer, all nodes who are descendants of X are also denoted buyers. This approach requires collaboration among system administrators to organize an optimal hierarchy. This is not suitable for a global environment which must scale to hundreds or thousands of machines.

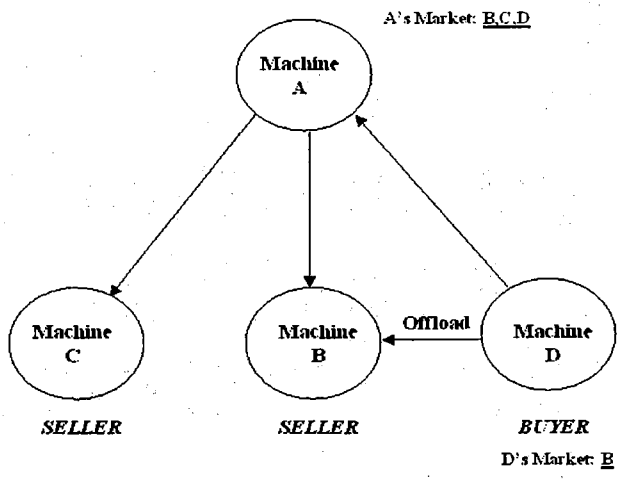


Figure 5.11. CPU Market Hierarchy.

We modify the hierarchical method, by allowing market initialization and registration to be bi-directional. Not only does the D-RTM register itself with machines designated by the system administrator, but machine also registers itself with the D-RTM. In such a situation, the market is organized by managers who are logically connected in a (complete) multidirectional topology. Because machines belong to more than one market, this configuration increases the communication overhead substantially. Communication increased from one message round to an expensive multicast. As shown in Figure 5.12, not only do machines B, C, and D notify machine A when buying or selling their CPU time, but, machine A must also notify machines B,C, and D when buying or selling its CPU time. Changes in the CPU status (i.e. Buyer/Seller), are notified to all machines within a market using a weak consistent replication strategy. We use weak consistent replication in order to reduce the communication over head. Notifications are replicated throughout the system by piggybacking the CPU status of the current machine along with Tasks that contain communication sends. For example: when an actor on machine B receives a communication from and actor on machine A, the CPU market on machine B is updated with the new CPU status of machine A. Although, machines are not instantly notified of a market change, use of this weak replication strategy provide eventual message delivery that is tolerated in our system [BN95].

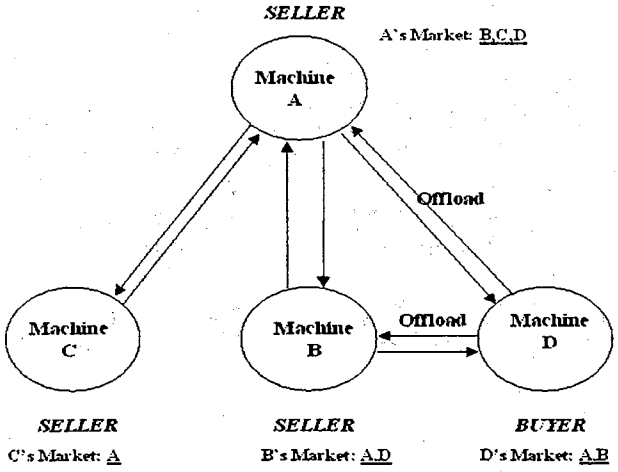


Figure 5.12. Host A Notifies Markets of B,C, and D.

Load Balancing Policy

Each machine within the distributed system maintains a data structure with information about the current machines within its market. These machines are denoted as buyers, or sellers. The load factor on the machine is relative to the number of threads currently running on the local machine. Other factors could also be used to determine the load. Such as: the total load on the machine, heuristic information, the actual CPU utilization, and the size of the computation. Most of these metrics are more complicated to determine. As shown in Figure 5.13, the Load Balancer maintains a load below 75% of the threshold, and 25% of the threshold above the minimum load (i.e. zero).

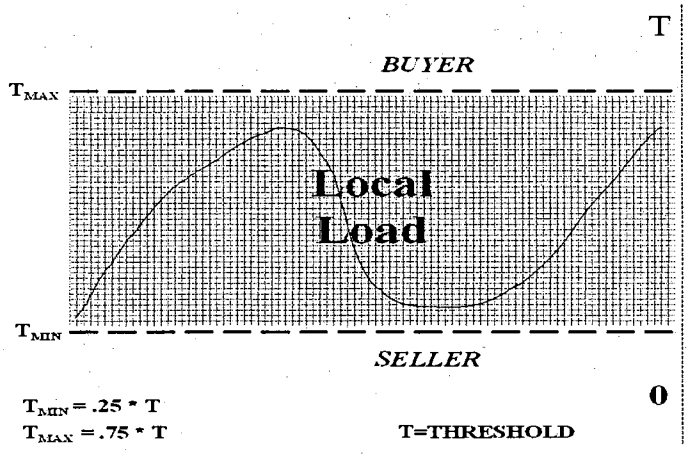


Figure 5.13. Load Balancing Policy.

Before starting a thread on the local machine, the load balancer checks the current load to insure that is within the threshold. If the load is not within the current threshold, the load balancer off-loads a local process to machines within its market who wish to sell their CPU (Figure 5.14). If there are no sellers within the market, the load balancer starts the process locally, and tries to off-load processes later. Note that the D-RTM is now a buyer of CPU time and needs to inform its market managers of its new status. We use a weak replication strategy by piggybacking the current status of a machine along with Tasks that contain communication sends. By default the status of a machine is *seller*. Therefore this field is changed to status *buyer*.

Load Balancing Algorithm:

Variable Definitions:

t : Task (communication sent throughout system)
load : Integer to denote the current load on the local machine
Threshold : Integer to denote the load limit on the local machine
BUYER,SELLER : constant to denote the state of the machine
CPUStatus : enumerator to denote the state of the machine (BUYER/SELLER)
host : contains the host location of an available CPU
scheduleLocal(t) : schedules the Task t (i.e. an actor) on the local machine
scheduleRemote(t, *host*) : schedules the Task t (i.e. an actor) at the location *host*
getAvailHost() : returns an available CPU (SELLER) from the market,
updateMarket(t) : update the *CPUStatus* of the machine from which the Task t originated

LoadBalancer :

1. Receive Task t
2. If t is an actor

   if $load + 1 \leq Threshold$, then
   set *CPUStatus* to **SELLER**
   scheduleLocal(t)
   increment *load* by 1
   Else
   set *CPUStatus* to **BUYER**
   host = getAvailHost()
   scheduleRemote(t, *host*)

   Else if t is a communication

   updateMarket(t)
   forward task to Message Handler
3. goto 1

Figure 5.14. Load Balancing Algorithm.

5.4 Logical Layer

The logical layer consists of the actual application specific computations that are executing on the local machines. The computation model consists of mobile actors which encapsulate: a behavior, an identity, a mail queue, and one thread (Figure 5.15). Each computation runs in its own thread, and may communicate with any other thread on the local/remote machines. Computations are expressed as Java programs using mobile actor semantics provided by constructs of the JMAS Mobile Actor API. The mobile actor API gives programmers the ability to create actors, change the state, or send communications to mobile actors within the global system. The underlying resources can be logically represented as mobile actors to build dynamic architecture topologies (Figure 5.16). This dynamic architecture gives the programmer an illusion of a global computer that can run

concurrent, distributed, and parallel applications. Implementation details of the underlying system are transparent to the programmer in the logical layer. A complete source listing of mobile actor programs is given in Appendix C.

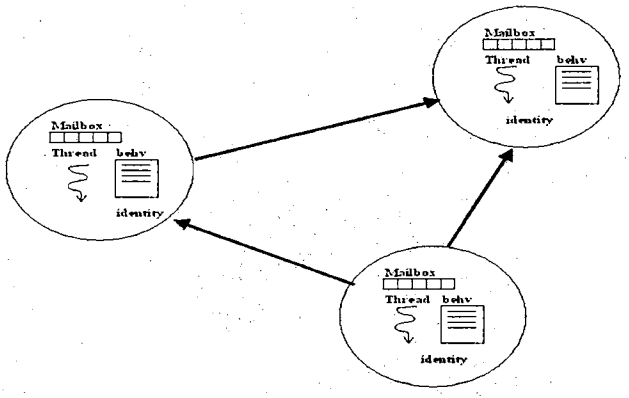


Figure 5.15. Computation Model.

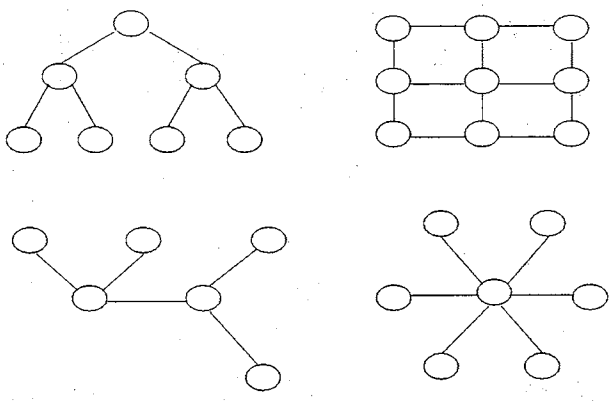


Figure 5.16. Logical View of Mobile Actor Architectures.

CHAPTER 6

PERFORMANCE EVALUATION

JMAS offers the basic infrastructure needed to integrate computers connected to the Internet into a distributed computational resource: an infrastructure for running coarse-grain parallel applications on several anonymous machines. Currently, cluster computing in a LAN setting are already being used extensively to run computation intensive applications. Through the efforts of GIMPS (Great Internet Mersenne Prime Search), a group of more than 700 workstations and PCs computed the 35th known Mersenne prime in 88 hours [Doc98]. In [Las97], the 48-bit RSA code was cracked using 3,500 workstations spread across Europe, and the 56-bit DES was cracked, using approximately 78,000 computers [DES97]. The examples above show that users are willing to participate in global computing efforts, and that there are applications that can be run very efficiently in wide area networks. In this section, we present results of the performance of our JMAS prototype. As shown in Figure 6.1, we conducted our experiments in a heterogeneous environment consisting of:

- 1 Sun MicroSystems Enterprise 3000, configured with two UltraSparc processors each running at 256MHz.
- 1 Sun Ultra Sparc workstations, configured with one 120 MHz processor.
- 1 Sun Sparc 5 workstation, configured with one 120 MHz processor.
- 14 Sun Sparc 20 workstations, each configured with one 200 MHz processor.
- 1 Sun Sparc 10 workstations, configured with one 166 MHz processor.

Each machine is connected by a 10 and 100 Mbit Ethernet. All experiments were conducted under the typical daily workloads. We tested each algorithm under a controlled environment of D-RTMs that were used strictly to run our experiments. CPU selection from the CPU market, was performed by the D-RTM using a round-robin selection policy. Under our controlled environment, an optimal

selection policy achieves the same results as round-robin CPU selection. We did not run our experiments using a random CPU selection policy. This was done to insure that all processes mapped to one and only one machine. In order to obtain a relative performance of our system, we calculate the average of the execution times over $N = 10$ experiments, producing an arithmetic mean (AM):

$$AM = \frac{1}{N} \sum_{1}^{N} Time_i$$

Where $Time_i$ is the execution time for the ith experiment. All experiments are compared with performance metrics obtained from similar computations on stand-alone workstations.

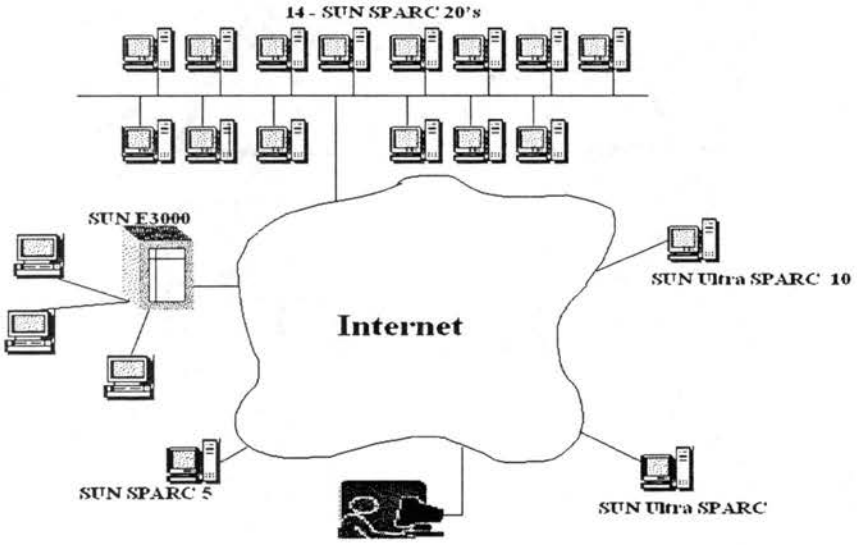


Figure 6.1. Test Environment.

6.1 Benchmarks

The overhead of migrating actors to remote locations and passing messages between remote actors are of great interest. We present experimental results for our prototype using two benchmarks: a Traveling Salesman application, and a Mersenne Prime application. We discuss their implementation and performance using the JMAS infrastructure.

6.2 Factors That Limit Speedup

A number of factors can contribute to limit the speedup achievable by a parallel algorithm executing in a global infrastructure such as JMAS. An obvious constraint is the size of the input program. If there is not enough work to be done by the number of processors available, then any parallel algorithm will not show an increase in speedup. Second, the number of process creations must be minimized. In particular, we are concerned with the creation of remote actors throughout the distributed system. Lastly, in a global environment were communication cost is high, the number and packet size of inter-process communications must be limited. Table 6.1 shows the performance of two micro-benchmarks to calculate the execution time for communication sends, and remote class loading using the JMAS prototype. A micro-benchmark is a small experiment used to monitor the performance of underlying system operations. Results were obtained using a test packet to send a communication, and load a Java class file between two machines.

Overhead	secs
Send	.006-.010
Remote Class Loading	.15-.28

Table 6.1. Micro benchmarks for a 10 Mbit Ethernet LAN using TCP sockets.

In general, the total cost of distributing a program for parallel execution is defined as:

$$T_{Cost} = Total_{loadTime} + Total_{commTime} + Total_{execTime}$$

Where $Total_{loadTime}$ is the time to load the needed Java class files to each machine within the system, $Total_{commTime}$ is the time spent sending communications between actors, and $Total_{execTime}$ is the time spent by all machines executing the fraction of the computation. Moreover, the total time to distribute the needed Java class files across N machines is:

$$Total_{loadTime} = (N-1) * t_{load}$$

Where t_{load} is the average time to load the needed Java class files to one machine within the system. We assume that the machines are organized using a master-slave topology. Such that, the master

is used to process a subcomputation, as well as, distribute $N-1$ subcomputations and receive the partial results from the other $N-1$ slave machines. Assuming we distribute the load evenly among N machines. Then the time to execute a fraction of the computation is:

$$Total_{execTime} = t_{seq}/N$$

Where t_{seq} is the total sequential execution time for the application. Given the load distribution above, if each subcomputation sends at most k messages, then the communication overhead $Total_{commTime}$ can be defined as:

$$Total_{commTime} = (N-1) * k * t_{send}$$

Where t_{send} is the average time to send a communication between two machines. Given N machines, we derive a general formula to define the total cost of distributing a program for parallel execution.

$$T_{Cost}(N) = (N-1) * t_{load} + (N-1) * k * t_{send} + t_{seq}/N \qquad Eq.1$$

We can estimate the performance of a given application using Equation 1. As shown below, in order to benefit from parallelization the following inequality must hold:

$$T_{Cost}(N) < t_{seq}$$

$$(N-1) * t_{load} + (N-1) * k * t_{send} + t_{seq}/N < t_{seq}$$

Solving the inequality, we find that the total cost (i.e. $T_{Cost}(N)$) is less than the sequential execution time (i.e. t_{seq}) for:

$$N < t_{seq}/(t_{load} + k * t_{send})$$

6.2.1 Remote Execution of Actors

As a mobile actor computation unfolds, mobile actors have the ability to implicitly navigate autonomously throughout the underlying network; causing the migration of code. On each of the experiments conducted in this chapter, we calculated the average time to load a Java class file over the network. On a standard 10 Mbit Ethernet network the time to load a remote class file ranges

between .15 and .28 seconds (Table 6.1). On average it takes .20 seconds to load a class file across the network. When considering distributing an application across several machines, one must take into consideration an upper bound on the amount of parallelism that can be exploited by distributing processes throughout a global system. In particular, we focus on the overhead associated with loading Java class files across the network (i.e. $Total_{loadTime}$). We can calculate the maximum number of machines p, needed to distribute the parallel computation without compromising the performance in speedup by finding the minimum execution time. The minimum execution time is the minimum value of the continuous function $T_{Cost}(p)$ on a closed bounded interval $[1, p]$; where $p = t_{seq}/(t_{load}+k*t_{send})$. We can simplify our calculation by assuming the communication overhead is minimal (i.e. $t_{send} = 0$). Giving the general formula for the total cost,

$$T_{Cost}(N) = (N-1) * t_{load} + t_{seq}/N$$

Taking the derivative of $T_{Cost}(p)$ with respect to p gives:

$$T'_{Cost}(p) = t_{load} - t_{seq}/p^2$$

Setting $T'_{Cost}(p) = 0$ and solving for p, gives

$$p = \lfloor\sqrt{t_{seq}/t_{load}}\rfloor \qquad Eq.2$$

Therefore, we can estimate the maximum speedup S as:

$$S = t_{seq}/T_{Cost}(p)$$

$$T_{Cost}(p) = 2 * t_{load}\sqrt{t_{seq}/t_{load}} - t_{load}$$

Giving,

$$S = \frac{2 * t_{seq}\sqrt{t_{seq}/t_{load}} + t_{seq}}{4 * t_{seq} - t_{load}} \qquad Eq.3$$

6.2.2 Message Passing

As stated in Chapter 5, communication in JMAS is asynchronous, reliable and connection-oriented. Messages between two actors, must be routed through a D-RTM on the local machine on which

the two actors reside. The Java Virtual Machine requires all communication to go through the Java network layer (i.e. *java.net*) and the complete TCP stack of the underlying OS. This causes a substantial software overhead compared to communication libraries of parallel machines. Using JMAS, a single message can be sent from one actor to another within .006-.010 seconds on a standard 10 Mbit Ethernet LAN (Table 6.1). As long as applications are coarse grained, the overhead of opening a socket connection can be ignored. Since message passing using Java TCP sockets is slow compared to dedicated parallel machines, and communication delays of large networks of heterogeneous machines is unpredictable, only computation-intensive parallel applications benefit from the JMAS infrastructure. In particular, we can estimate the performance of a parallel computation given the total communication overhead (i.e. $T_{commTime}$), and the total overhead (i.e. $T_{loadTime}$) associated with loading Java class files throughout the network. If k and t_{send} are constant, then the communication overhead (i.e. $T_{commTime}(N) = (N-1) * k * t_{send}$) is a linear function of N; where N denotes the total machines used. We can calculate the maximum number of machines p needed to distribute the parallel computation by taking the derivative of Equation 1 with respect to N; where $N = p$. Giving,

$$T'_{Cost}(p) = t_{load} + k * t_{send} - t_{seq}/p^2$$

Setting $T'_{Cost}(p) = 0$ and solving for p, gives

$$p = \lfloor \sqrt{t_{seq}/(t_{load} + k * t_{send})} \rfloor \qquad Eq.4$$

Therefore, we can estimate the maximum speedup S as:

$$S = t_{seq}/T_{Cost}(p)$$

$$T_{Cost}(p) = 2 * t_{load} \sqrt{t_{seq}/(t_{load} + k * t_{send})} - t_{load}$$

Giving,

$$S = \frac{2 * t_{seq} \sqrt{t_{seq}/(t_{load} + k * t_{send})} + t_{seq}}{4 * t_{seq} - (t_{load} + k * t_{send})} \qquad Eq.5$$

In the next two sections, we present the experimental applications used for performance evaluation.

6.3 Traveling Salesman Problem

Our first application is a parallel solution to the Traveling Salesman Problem (TSP). The Traveling Salesman Problem is as follows: given a list of n cities along with the distances between each pair of cities. The goal is to find a tour which starts at the first city, visits each city exactly once and returns to the first city, such that the distance traveled is as small as possible. This problem is known to be NP-complete (i.e. no serial algorithm exists that runs in time polynomial in n, only in time exponential in n), and it is widely believed that no polynomial time algorithm exists. In practice, we want to compute an approximate solution, i.e. a single tour whose length is as short as possible, in a given amount of computation time.

More formally, we are given a graph $G = (N, V, W)$ consisting of a set N of n nodes (or cities), a set of edges $V = (i, j)$ connecting cities, and a set of nonnegative weights $W = w(i, j)$ giving the length of edge (i, j) (i.e. the distance from city i to city j). The graph is directed, so that an edge (i, j) may only be traversed in the direction from i to j, and edge (j, i) may or may not exist. Similarly, $w(i, j)$ does not necessarily equal $w(j, i)$, if both edges exist.

There are a great many algorithms for this important problem, some of which take advantage of special properties like symmetry (edges (i, j) and (j, i) always exist or do not exist simultaneously, and $w(i, j) = w(j, i)$) and the triangle inequality ($w(i, j) <= w(i, k) + w(k, j) \forall i, j, k$). In this application we assume none of these properties hold. For simplicity, though, we assume all edges (i, j) exist, and all $w(i, j)$ are positive integers (note that setting some $w(i, j)$ to be very large effectively excludes it from appearance in a solution).

6.3.1 TSP Algorithm

We take a naive approach to solving the TSP using an Exhaustive-Search. The exhaustive-search algorithm searches all $(n-1)!$ possible paths, while keeping the best path searched so far. We generate all possible paths using a *Permutation*() function on the number of cities n. The permutation function generates a lexicographical ordering of all possible paths. We divide the permutations equally among a set of processors p; such that each processor searches $(n-1)!/p$ possible paths

(Figure 6.2). Processors are arranged in a master-slave design. A complete source code listing of the TSP solution using mobile actors is given in Appendix C.6.

Variable Definitions:

n : Integer to denote the number of cities
p : Integer to denote the number of machines
mintour : Integer to denote the permutation of the best tour searched
start : Integer to denote the starting permutation in lexicographical order
stop : Integer to denote the ending permutation in lexicographical order
resultTour : Integer to denote the best tour search for a specified range lexicographically
itself : Actor address of itself
cust : Actor address to send result
range : Integer to denote the total permutations (tours) to check
Permutation(i) : Generates the ith tour in lexicographical order

behavior Master :

1. *mintour* = 0
2. *range* = $(n-1)!/p$
3. for each processor i : 1 to $p-1$ do
   create a Remote actor assume behavior *Slave*, return address of actor as x
   send *start = (i*range), stop = ((i+1)*range)*, and the address of *itself* to x
4. **become** *itself* and wait for p results
5. for i : 1 to p do
   receive *resultTour*
   if Permutation(*resultTour*) distance $\leq$ Permutation(*mintour*) distance
   set *mintour* to *resultTour*

behavior Slave :

1. recv *start*, *stop*, and address of *cust* to send result
2. *mintour* = *start*
3. for i equal *start* to *stop* do
   if Permutation(i) distance $\leq$ Permutation(*mintour*) distance
   set *mintour* to i
4. **send** *mintour* to *cust*

Figure 6.2. TSP Algorithm.

6.3.2 Measurements

In order to complete our set of measurements in a reasonable amount of time we chose to test our TSP solution primality for $N = \{4, 5, 10, 13\}$ cities. We conducted the experiment in an environment consisting of up to 15 machines, and compared the results with a sequential application running on a SPARC 20 workstation. As shown in Figure 6.3, there is no significant gain in performance for $N < 10$. This is due to the overhead associated with loading Java class files across the network. Figure 6.4 displays the execution time of a TSP solution for $N = 5$ versus its remote Java class loading time. As the number of machines p increase, the load time increases, causing the execution

time to increase; exceeding the execution time for a sequential solution. Notice we achieve the best performance for $p = 4$ machines. For $N \geq 10$, our TSP solution gives better performance than serial execution. In particular, for $N = 13$ the speedup obtained is close to linear. Due to limited resources, we were unable to test the scalability of the application for large values of p. We estimate the performance of our TSP application using Equations 1,2 and an average load time $t_{load} = .15$ secs. As illustrated in Table 6.2, the average CPU utilization for the best possible number of machines p is 50%. As the number of processors p approach $(N-1)!$, the speedup obtained will decrease significantly; due to under utilization of processors and the overhead associated with loading Java class files across the network (Figure 6.5). The estimates are also reflected in Figure 6.3. These results show that our framework is well suited for course grain applications. The TSP application also scales well to large computation sizes (Figure 6.6).

Prob. Size	t_{seq} secs	Max. p	Max. S	Utilization
N=5 Cities	3.007	4	2.24	56%
N=10 Cities	24.441	12	6.33	52.7%
N=13 Cities	36655.848	494	247.42	50%

Table 6.2. Estimating the Performance of TSP.

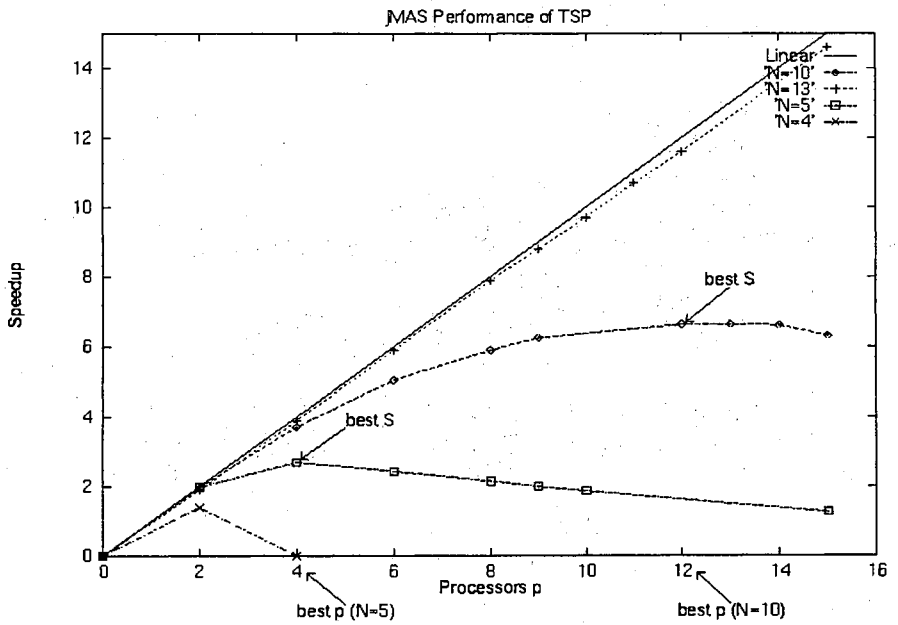


Figure 6.3. Speedup of TSP.

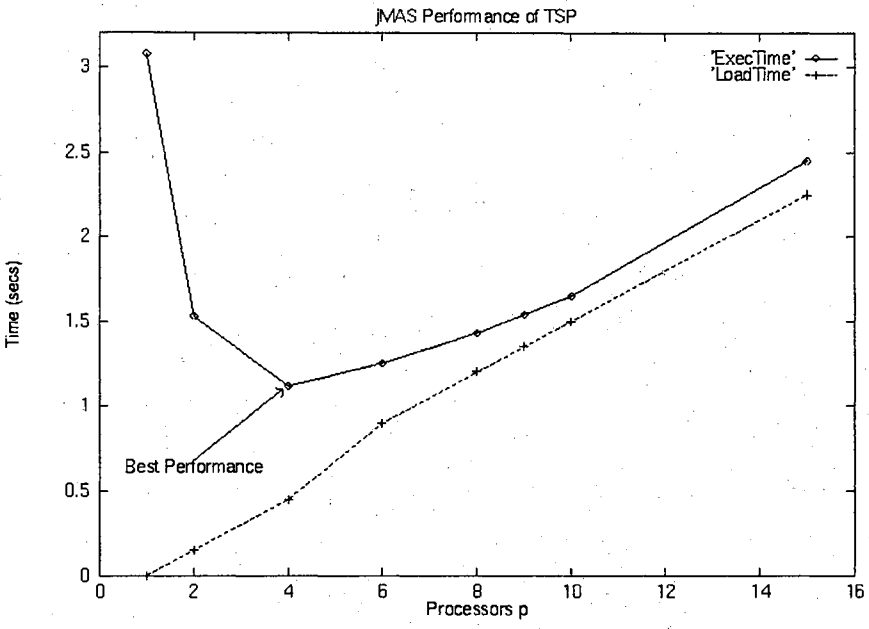


Figure 6.4. Execution Time vs Load Time.

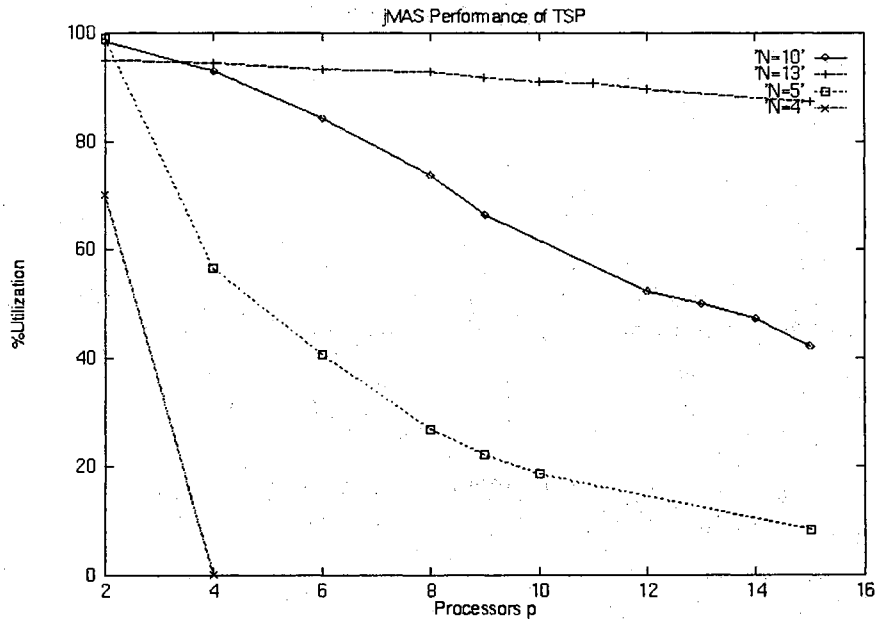


Figure 6.5. CPU Utilization of TSP.

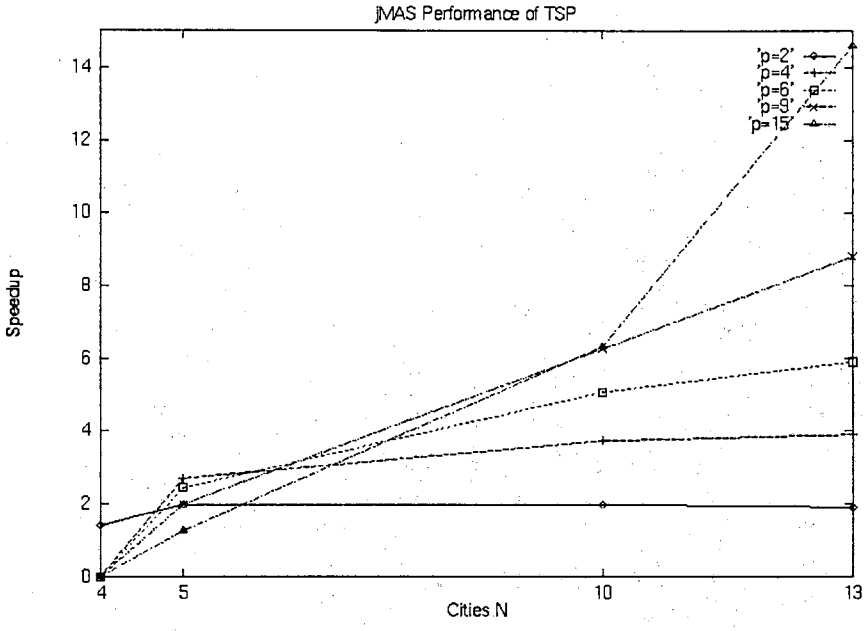


Figure 6.6. Scalability of TSP.

6.4 Mersenne Prime Application

For our second application, we implemented a parallel primality test which is used to search for Mersenne prime numbers [Doc98]. This type of application is well suited for our infrastructure. It is very coarse grained with low communication overhead.

A Mersenne prime is a prime number of the form $2^p - 1$, where the exponent p itself is prime. These are traditionally the largest known primes. Encryption and decryption methods are typical applications which utilize large prime numbers. Searching and verifying Mersenne primes using computer technology has been conducted since 1952 [Doc98]. To date 37 Mersenne primes have been discovered. Only up to the 35th Mersenne prime has been verified. The current record holder is $2^{1398269} - 1$ and was discovered through the use of over 700 PCs and workstations worldwide. With larger and larger prime exponents, the search for Mersenne primes becomes progressively more difficult.

6.4.1 Mersenne Prime Algorithm

In our implementation, each prime is tested based on the following theorem:

> **Lucas-Lehmer Test:** For p odd, the Mersenne number $2^p - 1$ is prime iff $2^p - 1$ divides $S(p-1)$; where $S(n+1) = S(n)^2 - 2$, and $S(1) = 4$. The proof can be obtained from [Doc98].

We develop a mobile actor program to test for Mersenne primality, given a range of prime numbers (Figure 6.7). Processors are arranged in a master-slave design. As shown below, our application works as follows:

> Given N machines and a range r of prime numbers, we divide the search such that:
>
> each machine tests for a Mersenne prime using the Lucas-Lehmer Test for a range of primes. Each range is of size r/N.

A complete source listing of the Mersenne Prime application is in Appendix C.7.

Variable Definitions:

r : Integer to denote the amount of primes to test
N : Integer to denote the number of machines
Lucas(x) : Performs Lucas-Lehmer test on x
itself : Actor address of itself
cust : Actor address to send result
range : Integer to denote the range of primes to check
start : Integer to denote the starting prime number
stop : Integer to denote the prime number used as a sentinel
$recv_{count}$: Integer to denote the total results received
PRIME : enumerator returned from Lucas(x); if x is a prime number
SINK : message to denote the termination of a subcomputation

behavior Master :

1. *range* $= r/N$
2. for each processor i : 1 to $N-1$ do
   create a Remote actor assume behavior *Slave*, return address of actor as x
   send *start = (i*range), stop = ((i+1)*range)*, and the address of *itself* to x
3. **become** *itself* and wait for N results
4. set $recv_{count} = 0$
5. receive result
6. if *result* is *SINK*
   increment $recv_{count}$ by 1
   Else
   print "$2^{result} - 1$ is PRIME!"
7. if $recv_{count} < N$, then goto 5

behavior Slave :

1. recv *start, stop*, and address of *cust* to send result
2. for i : *start* to *stop* do
   if Lucas(i) is **PRIME**
   send i to *cust*
3. **send** *SINK* to *cust*

Figure 6.7. Mersenne Prime Algorithm.

6.4.2 Measurements

For our measurements, we chose to test the Mersenne primality for all exponents between 4000 and 5000. Known primes within this range are $2^{4253} - 1$ and $2^{4423} - 1$. The reason for selecting this range is that:

1. we tried to make the number large enough to simulate the true working conditions of the application,

2. we wanted to keep them small enough to be able to complete our set of measurements in a reasonable amount of time.

We conducted the experiment in an environment consisting of up to 15 machines, and compared the results with a sequential application running on a SPARC 20 workstation. As shown in Figure 6.8, our application scales to 15 machines linearly. The speedup obtained is slightly lower than linear speedup. This is because we decompose the range of primes to be tested unevenly in terms of the amount of work to be done.

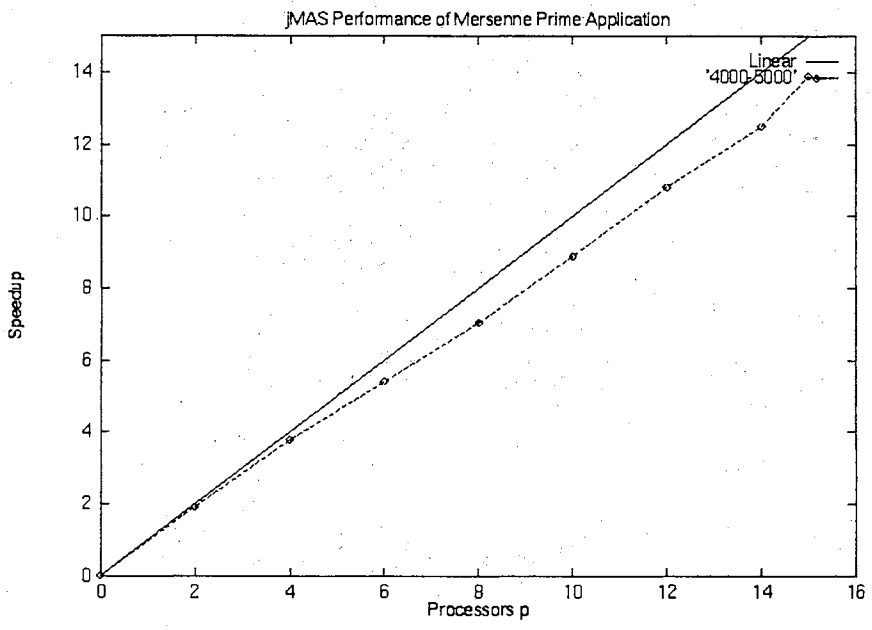


Figure 6.8. Speedup of Mersenne Prime.

For instance, testing if $2^{4000}-1$ is prime, can be done much faster than testing if $2^{4999}-1$ is prime. We split the ranges in groups such that, the last machine receives the last group consisting of the largest numbers. Due to limited resources, we were unable to test the scalability of the application for large values of p. We estimate the performance of the Mersenne Prime application using Equations 1,2; where the average load time $t_{load} = .20$ secs, and the average sequential execution time $t_{seq} = 83432$ secs. As shown in Table 6.3, results show that the application scales up to 646 machines with an overall speedup of 323. From our results we can assume that for $p > 646$, the range of primes to test decreases causing under utilization of CPUs (Figure 6.9). Also, for every new machine added, the time to load Java class files increases causing a decrease in performance.

Application	t_{seq} secs	Max. p	Max. S	Utilization
Mersenne Prime	83432	323	646	50%

Table 6.3. Estimating the Performance of the Mersenne Prime Test.

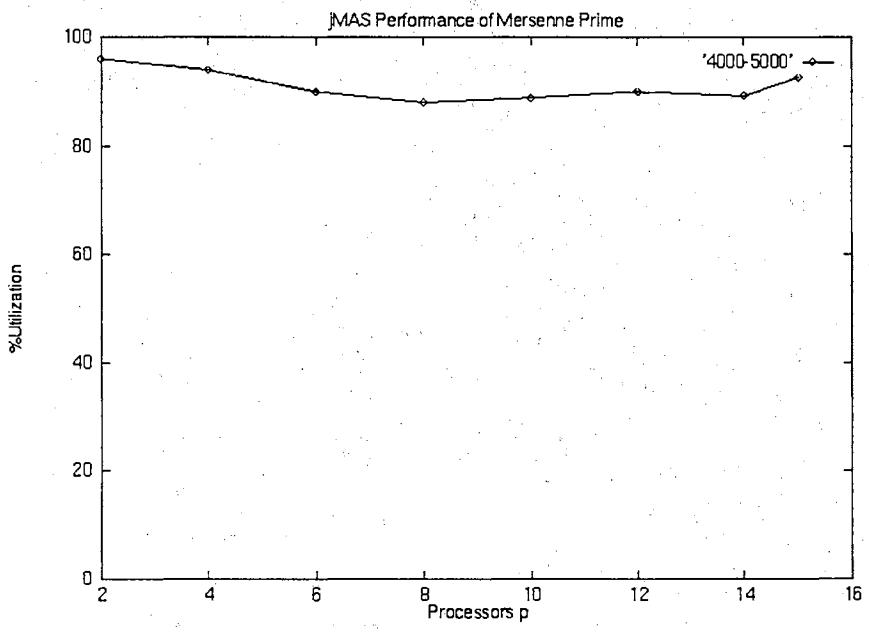


Figure 6.9. CPU Utilization of Mersenne Prime.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis we introduce *mobile actors*; a parallel programming paradigm for distributed parallel computing based on mobile agents and the *actor* message passing model [Agh86]. The Actor-based message passing model supports dynamic architecture topologies that make it ideal for distributed parallel computing. We implement a prototype system (*JMAS*) based on the mobile actor model using Java technology. JMAS is a globally distributed computing environment for executing mobile actor computations. JMAS is designed using Java technology [Inc95], and requires a programming style different from commonly used approaches to distributed computing. JMAS allows a programmer to create mobile actors, initialize their behaviors, and send them messages using constructs provided by the JMAS Mobile Actor API. As the computation unfolds, mobile actors have the ability to implicitly navigate autonomously throughout the underlying network. New messages are generated, new actors are created, and existing actors undergo state change. We evaluate the performance of our system using two benchmarks: a Mersenne Prime Application, and the Traveling Salesman Problem. The degree of parallelism obtained from distributing mobile actors throughout the system is limited due to the overhead associated with migrating Java class files, and the amount of inter-process communication. In particular, we are bound by the number of processors

$$p = \mathrm{O}(\lfloor\sqrt{t_{seq}/(t_{load} + k * t_{send})}\rfloor)$$

to distribute the parallel computation; where t_{seq} is the sequential execution time of the application, t_{load} is the average time to load the needed Java class files to one machine, k is the total message rounds sent per machine, and t_{send} is the average time to send a communication between two machines. Given p we can estimate the speedup S as:

$$S = t_{seq}/T_{Cost}(p)$$

When the enhanced performance using p machines, is denoted as a general formula

$$T_{Cost}(p) = (p-1) * t_{load} + (p-1) * k * t_{send} + t_{seq}/N$$

the speedup is,

$$S = \frac{2 * t_{seq}\sqrt{t_{seq}/(t_{load} + k * t_{send})} + t_{seq}}{4 * t_{seq} - (t_{load} + k * t_{send})}$$

Our estimates for the TSP and Mersenne Prime applications, show that each application scales to large numbers of machines N. But for $N > p$, we estimate a decrease in performance; due to the under utilization of CPUs, and the significant overhead associated with loading the needed Java class files and sending communications throughout the system. These results show that our framework is well suited for course grain applications.

7.2 Future Work

In order to improve the performance of Java programs for high performance computing, the execution time for interpreting Java bytecode must be addressed. Performance boosters such as a JIT (just in-time) Java compiler provide a more efficient execution of Java applications [Inc95]. Some JIT compilers give increases in performance that match the execution of optimized C compiliers. We suggest the incorporation of a JIT Java compiler to improve the performance of JMAS. Issues such as fault tolerance and security need to be addressed and implemented within the JMAS framework. High-level communication abstractions should be addressed within the JMAS Mobile Actor API. Examples are barrier actors, mutex actors, call/return communication, and actorSpaces [AP91]. Distributed I/O, exception handling, and the intergration of mobile actors and sequential applications using graphical user interfaces, need to be addressed. The prototype also needs to be tested on machines running Microsoft Windows NT and 95 operating systems. Providing support for connection-less communication using UDP sockets could improve the performance of actor communication, as opposed to TCP sockets which involve communication overhead to establish and maintain a reliable connection. A connection-less communication system would be useful in scalable computing clusters, or networks of workstations. Lastly, more work needs to be done to provide gen-

eral formulas for estimating the performance of parallel compuations distributed among a network of machines unevenly.

BIBLIOGRAPHY

[ACP95] T. Anderson, D. Culler, and D. Patterson. A case for now (network of workstations). In *IEEE Microcomputer*. IEEE, 1995.

[Age97] AgentSoft. *AgentSoft's products: LiveAgent and SearchAgent.* http://www.agentsoft.com, 1997.

[Agh86] G. Agha. *Actors: A model of concurrent computation in distributed systems.* M.I.T. Press, 1986.

[Agh89] Gul Agha. Supporting multiparadigm programming on actor architectures. In *Proceedings of Parallel Architectures and Languages Europe*, pages 1–19. LNCS, 1989.

[AHP91] Gul Agha, Chris Houck, and Rajendra Panwar. Distributed execution of actor programs. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing.* Santa Clara, 1991.

[AKP90] G. Agha, W. Kim, and R. Panwar. Actor languages for specification of parallel computations. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 00, 1990.

[AMST91] Gul Agha, I. Mason, S. Smith, and C. Talcott. Towards a theory of actor computation. In *3rd International Conference on Concurrency Theory CONCUR '92*, pages 565–579. LNCS, 1991.

[AP91] Gul Agha and R. Panwar. An actor-based framework for heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 21, 1991.

[AS88] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 2:9–23, 1988.

[aSU97a] NPAC at Syracuse University. *Computing on the Web: New Approaches to Parallel Processing Petaop and Exaop Performance in the Year 2007.* Online Technical Report, http://www.npac.syr.edu/users/gcf/petastuff/petaweb/, 1997.

[aSU97b] NPAC at Syracuse University. *WebFlow: A Visual Programming Paradigm for Web and Java Based Coarse Grain Distributed Computing.* Online Technical Report, http://www.npac.syr.edu/projects/javaforcse/cpande/sufurm.ps, 1997.

[atoSM96a] JavaSoft a trademark of Sun Microsystems. *Java Object Serialization.* http://chatsubo.javasoft.com/current/serial/index.html, 1996.

[atoSM96b] JavaSoft a trademark of Sun Microsystems. *Java Remote Method Invocation (RMI).* http://chatsubo.javasoft.com/current/rmi/index.html, 1996.

[BBB96] J. Baldeschwieler, R. Blumofe, and E. Brewer. Atlas: An infrastructure for global computing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for WorldWide Applications.* ACM SIGOPS, 1996.

[BFD96] L. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 18:55–61, 1996.

[BG98] L. Burge and K. George. An actor based framework for distributed mobile computation. In *PDPTA - Parallel Distributed Processing Techniques and Applications.* CSREA, 1998.

[BJK+95] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN*, pages

207–216. Symposium on Principles and Practice of Parallel Programming (PPoPP), 1995.

[BKKW96] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems.* PDCS, 1996.

[BL96] T. Berners-Lee. Www: Past, present and future. *IEEE Computer*, 18:69–77, 1996.

[BN84] A. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.

[BN95] L. Burge and M. Neilsen. Variable-rate timestamped anti-entropy. In *ISMM International Conference on Parallel and Distributed Computing and Systems.* 7th IASTED, 1995.

[BN96] L. Burge and M. Neilsen. A decentralized algorithm for communication efficient distributed shared memory. In *11th Annual Symposium on Applied Computing - Distributed and Parallel Processing.* SAC, 1996.

[BSST96] R. Brecht, H. Sandhu, M. Shan, and J. Talbot. Paraweb: Towards world-wide supercomputing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for WorldWide Applications.* ACM SIGOPS, 1996.

[BVN91] F. Baude and G Vidal-Naquet. Actors as a parallel programming model. In *Proceedings of the 8th Symposium on theoretical Aspects of Computer Science*, page 480. LNCS, 1991.

[CA91] C. Callsen and G. Agha. Open heterogeneous computing in actorspace. *Journal of Parallel and Distributed Computing*, 21:289–300, 1991.

[CDL+96] K. Chandy, B. Dimitron, H. Le, J. Mandleson, M. Richardson, A. Rifkin, P. Sivilotti, W. Tawaka, and L. Weisman. A world-wide distributed system using java and the internet. In *Proceedings of the 5th IEEE Internation Symposium on High Performance Distributed Computing.* IEEE HPDCS, 1996.

[Cli81] W. D. Clinger. Foundation of actor semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

[CLNR97] N. Camiel, S. London, N. Nisan, and O. Regen. The popcorn project: Distributed computation over the internet in java. In *Proceedings of the 5th Internation World Wide Web Conference.* W3, 1997.

[CM96] W. Chang and D. Messerschmitt. Dynamic deployment of peer-to-peer networked applications to existing world wide web browser. *Proceedings of the Telecommunications Information Network Architectures (TINA)*, 1, 1996.

[Con96] WWW Consortium. *Jigsaw Web Server.* http://www.w3.org/pub/www/jigsaw, 1996.

[DD96] H.M Deitel and J. P. Deitel. *Java How to Program.* Printice Hall, 1996.

[DES97] DESCHALL. Internet-linked computers challenge data encryption standard. Technical report, Press Release, 1997.

[Doc95] Online Document. *Mobile Agents: Are they a good idea?* http://www.eit.com /goodies/list/www.lists/www-talk.1995q1/0764.html, 1995.

[Doc98] Online Document. *Mersenne Primes: History, Theorems and Lists.* http://www.utm.edu/research/primes/mersenne.shtml, 1998.

[DoCS96a] University of California at Santa Barbara Dept. of Computer Science. *Javalin:*

Internet-Based Parallel Computing Using Java. Online Technical Report, http://www.cs.ucsb.edu/ danielw/Papers/wjsec97.ps, 1996.

[DoCS96b] University of Maryland College Park Dept. of Computer Science. *Network-aware Mobile Programs.* Online Technical Report, http://www.cs.umd.edu/TR/CS-TR-3659, 1996.

[DoCS97a] Old Dominion University Dept. of Computer Science. *Web Based Framework for Distributed Computing.* Online Technical Report, http://www.cs.odu.edu/ techrep /techreports/TR_97_21.ps.Z, 1997.

[DoCS97b] Rice University Dept. of Computer Science. *Java/DSM: A Platform for Heterogeneous Computing.* Online Technical Report, http://www.cs.rice.edu/ weimin/papers /java97.ps, 1997.

[DoCS97c] University of California at Santa Barbara Dept. of Computer Science. *SuperWeb: Research Issues in Java-Based Global Computing.* Online Technical Report, http://www.npac.syr.edu/projects/javaforcse/cpande/UCSBsuperweb.ps, 1997.

[fDRC94] L. Cardelli for DEC Research Center. *Obliq: A language with distributed scope.* http://www.research.digital.com/SRC/Obliq/Obliq.html, 1994.

[FF96a] G. Fox and W. Formaski. Towards web/java based high performance distributed computing - and evolving virtual machine. In *Proceedings of the 5th IEEE Internation Symposium on High Performance Distributed Computing.* IEEE HPDCS, 1996.

[FF96b] G. Fox and W. Furmanski. Towards web/java based high performance distributed computing - an evolving virtual machine. In *Conference on High Performance Distributed Computing*, page 10. IEEE, 1996.

[FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1, 1997.

[GK92] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with piranha. In *Proceedings of the 6th ACM International Conference on Supercomputing.* ACM, 1992.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[Gra95] R. Gray. Agent tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents.* Fourth International Conference on Information and Knowledge Management (CIKM '95), 1995.

[GWtLT97] A. Grimshaw, W. Wulf, and the Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 20:39–45, 1997.

[Ham96] M. Hamilton. Java and the shift to net-centric computing. *IEEE Computer*, 1:31–39, 1996.

[Hay88] J. P. Hayes. *Computer Architecture and Organization.* McGraw-Hill, Inc, 1988.

[HBB96] D. Halls, J. Bates, and J. Bacon. *TUBE: Flexible Distributed Programming Using Mobile Code.* http://www.cl.cam.ac.uk/users/dah28/position-final/position-final.html, 1996.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8:323–364, 1977.

[Inc94] Oracle Inc. Oracle press release. Technical report, Oracle, 1994.

[Inc95] Sun Microsystems Inc. *The Java Virtual Machine Specification.* Online Technical Report, http://java.sun.com, 1995.

[Inc97] JavaSoft Inc. *Jeeves Java-based Web Server.* Online Technical Report, http://www.javasoft.com/javastore/jserv, 1997.

[KAB98] A. Keren and Institute of Computer Science Hebrew University A. Barak. *Parallel Java Agents.* http://cs.huji.ac.il/, 1998.

[KBW97] L. Kale', M. Bhandarkar, and T. Wilmarth. Design and implementation of parallel java with global object space. In *PDPTA International Conference*, pages 235–244. PDPTA, 1997.

[KCDZ94] P. Kelcher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of USENIX.* USENIX, 1994.

[KKT92] H. Koch, L. Krombholz, and O. Theel. *A Brief Introduction into the World of Mobile Computing (extended abstract).* http://www.tm.informatik.th-darmstadt.de/, 1992.

[KZ97] J. Kiniry and D. Zimmerman. A hands-on look at java mobile agents. *IEEE Internet Computing*, 1:21–29, 1997.

[Las97] A. Lash. *48-bit crypto latest to crack. C-NET: The Computer Network.* http://www.news.com/News/Item/0,4,7849,4000.html, 1997.

[LDD96] A. Lingnau, O. Drobnik, and P. Domel. *An HTTP-based Infrastructure for Mobile Agents.* http://www.tm.informatik.uni-frankfurt.de/agents/www4-paper.html, 1996.

[LI97] Argonne National Laboratory and USC Information Science Institute. *The Nexus Multithreaded Runtime System.* http://www.mcs.anl.gov/nexus, 1997.

[LL88] M. Litzkow and M. Linwy. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems.* ICDCS, 1988.

[MC96] Microelectronics and Computer Consortium. *Rosette Reference Manual.* http://www.mcc.com/projects/carnot/rosette, 1996.

[Obj96] ObjectSpace. *Voyager.* http://www.objectspace.com/Voyager, 1996.

[oCaI96] University of California at Irvine. *MESSENGERS.* Online Technical Report, http://www.ics.uci.edu/ bic/messengers, 1996.

[oCS96] Sewdish Institute of Computer Science. *AKL, AGENTS, and Penny.* http://www.sics.se/ps/agents.html, 1996.

[oMCS97] Dept. of Math and Emory University Computer Science. *IceT: Distributed Computing and Java.* Online Technical Report, http://www.mathcs.emory.edu/ gray/, 1997.

[oT96] University of Tromse's. *Tacoma.* Online Technical Report, http://www.cs.uit.no/DOS/Tacoma, 1996.

[OW97] S. Oaks and H. Wong. *Java Threads.* O'Reilly, 1997.

[Pei97] H. Peine. *Ara - Agents for Remote Action.* http://www.uni-kl.de/AG-Nehmer/Ara/, 1997.

[Rey97] F. Reynolds. Evolving an operating system for the web. *IEEE Computer*, 1:90–92, 1997.

[Rie94] Doug Riecken. M: An architecture of distributed agents. *Communications of the ACM*, 37:48–53, 1994.

[rL96] Tokyo research Lab. *Aglet Workbench.* http://www.trl.ibm.co.jp, 1996.

[SaOGI97] Computer Science and Engineering at Oregon Graduate Institute. *MIST.* http://www.cse.ogi.edu/DISC/projects/mist/overview.html, 1997.

[SH97] M. Singh and M. Huhns. Internet-based agents: Applications and infrastructure. *IEEE Internet Computing*, 1:8–9, 1997.

[Sta84] J. A. Stankovic. A perspective on distributed computer systems. *IEEE Transactions on Computers*, 33:28–41, 1984.

[Sun90] V. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2, 1990.

[SVN91] Y. Sami and G. Vidal-Naquet. Formalization of the behavior of actors by colored peri nets and some applications. In *Proceedings of Parallel Architectures and Languages Europe*. PARLE, 1991.

[Tan92] A. Tanenbaum. *Modern Operating Systems*. Englewood Cliffs, N. J. : Prentice Hall, 1992.

[Tan96] A. Tanenbaum. *Computer Networks*. Englewood Cliffs, N. J. : Prentice Hall, 1996.

[TMN98] H. Takagi, S. Matsuoka, and H. Nakada. *Ninflet: A Migratable Parallel Object Framework using Java*. http://ninf.etl.go.jp/, 1998.

[UoCaI96] Dept. of Computer Science University of California at Irvine. *MESSENGERS: A Distributed Computing Environment for Autonomous Objects*. Online Technical Report, http://www.ics.uci.edu/ bic/messengers, 1996.

[Van97] L. Vanhelsuwe. *Create your own supercomputer with Java*. http://www.javaworld.com/javaworld/jw-01-1997/jw-01-dampp.html, 1997.

[Ven97] Bill Venners. *Solve real problems with aglets, a type of mobile agent*. http://www.javaworld.com/javaworld/jw-05-1997/jw-05-hood.html, 1997.

[Whi94a] J. White. Mobile agents make a network an open platform for third party developers. *IEEE Computer*, 27:89–90, 1994.

[Whi94b] J. White. Telescript technology: The foundation for the electronic marketplace, general magic white paper. Technical report, General Magic, 1994.

APPENDIX A

JMAS: INSTALLATION AND USER GUIDE

A.1 Setting up JMAS on your System

JMAS is distributed as both source and a zipped class library and may be downloaded from the the following URL: *http* : *//a.cs.okstate.edu/* ~ *blegand/JMAS*. JMAS is written entirely in Java. This release has been developed under JDK1.1-3. Due to time constraints it has only been tested on Sun Solaris. Future testing will include LINUX and Win95/NT. Unless you plan on re-compiling from the source, setting up JMAS is easy. First, unpack the distribution somewhere convenient. This should create the following tree of directories

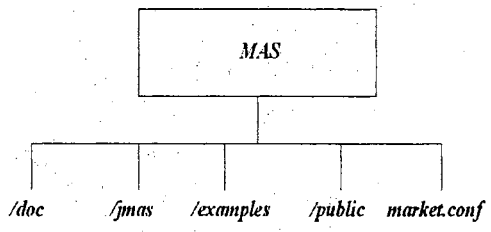


Figure A.1. JMAS Directory Structure.

If you plan on running JMAS from the zipped binaries (JMAS.zip), they are located in the top level directory. Just include the path of location of the *MAS* directory in the environment variable *JMASPATH*. Next, include the JMAS.zip file in your CLASSPATH. If you plan on recompiling from the source, add the location of the *MAS* directory to your CLASSPATH. For example, if I unpacked the distribution in my home directory. To work from the zipped binaries, my JMASPATH and CLASSPATH might be set as follows:

$$JMASPATH = \$HOME/MAS$$

$$CLASSPATH = \$JAVAHOME/lib/classes.zip : \$JMASPATH/jmas.zip$$

To work from the recompiled source, my JMASPATH and CLASSPATH would be:

$$JMASPATH = \$HOME/MAS$$

$$CLASSPATH = \$JAVAHOME/lib/classes.zip : \$JMASPATH/jmas$$

Make sure that your CLASSPATH includes the default Java packages as the first entry in the CLASSPATH.

A.2 Starting the JMAS D-RTM

Before starting JMAS take note that there exists a directory *public* in the top level directory. All executables written using JMAS MUST be put into this directory. The directory is a repository where remote actors can locate publicly available behaviors on your system. You must also denote any other machine you would like in your market cluster. This information is stored in the *market.conf*, which resides also in the top level directory. Make sure that the machines you list are also running JMAS. In order to change the current threshold value for your machine, modify the second parameter within the file *jmasd* in the $\$JMASPATH/bin$ directory.

To start JMAS, issue the command **jmasd** [*-i*] at the command prompt; where the *-i* is optional. To run with the GUI specify the *-i* option. Once JMAS is up and running you should see a graphical user interface which displays information about the current processes running on the local machine, the current CPU market, and a Threshold meter which displays the current load (Figure A.2).

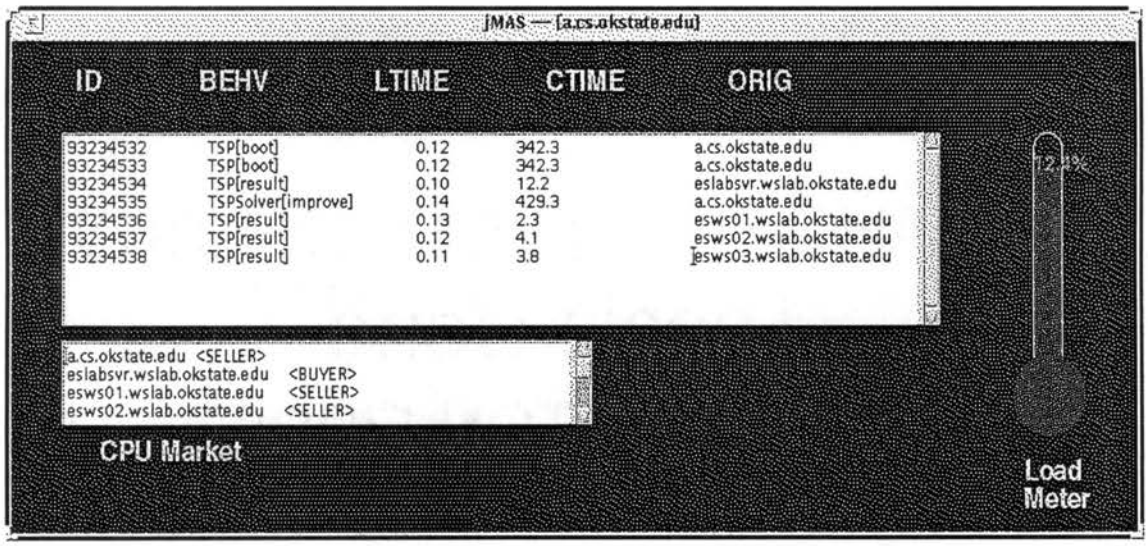


Figure A.2. JMAS Graphical User Interface.

A.3 Terminating the JMAS D-RTM

JMAS can be terminated through the GUI (i.e. closing the GUI window), or by issuing a UNIX system command. Because the JMAS D-RTM runs in the background, a process id must be known in order to kill the process. For UNIX machines this is obtained using the *ps* command. To kill the process, at command prompt issue:

% kill -9 < *id* >

A.4 Compiling Mobile Actor Programs

Compilation of mobile actor programs follows the same procedure for compilation of standard Java applications (i.e. *javac*). Programs may be debugged using the standard Java debugger (i.e. *javab*). Programs may be written as a Java application (i.e. with a *main*()), or as an mobile actor application (i.e. using a *boot*() method). Writing a program using a *boot*() method will be discussed in a future section. **NOTE:** JMAS is not aware of program modifications when recompiling programs. Therefore, the system must be rebooted in order to remove the last copy of the Java class from the system cache. Future versions of JMAS will provide a system command to purge the local cache during recompilation.

A.5 Executing Mobile Actor Programs

There are several methods you can use in order to execute actor programs. JMAS can invoke actor programs that are boot strapped with a *boot()* method, or you can write a Java program that instantiates an actor object for use. Invoking actor programs through their boot method is very simple. Given the code provided in the Mobile Actor API (Appendix B.1.1), we can invoke the HelloWorld program by issuing the following command at the system prompt:

% jmas HelloWorld

Process status information can be obtained by issuing the same command with the $-ps$ option. It can also be viewed visually through the *jmasd* graphical user interface (Figure A.2). Issuing the

command *jmas −ps* produces the following result.

```
-------------------------------------------------------------------------------

Process status for [a.cs.okstate.edu] <2.0% Load>: 2 Jobs...


id              behv                lTime          cTime          orig.

932343411       HelloWorld[boot]    0.12            4.0       a.cs.okstate.edu

932343412       WorldActor[world]   0.10            10.2      a.cs.okstate.edu

-------------------------------------------------------------------------------
```

Note: All code must be placed within the public directory. As shown below, in order to integrate actors within a java application , all you need to do is create an instance of the actor or mobile actor object. Because actors communicate using asynchronous messages there is no way to obtain the result back to the main. This feature will be added in future versions of JMAS.

```
public class StartActor
{
        public static void main(String args[])
        {
                MobileActor  ma = new MobileActor();

                ActorAddress ha = ma.createRemote("HelloActor","a.cs.okstate.edu");
                ma.send(ha,"hello");
        }
}
```

APPENDIX B

JMAS: MOBILE ACTOR API SPECIFICATION

This section provides the interface to the Mobile Actor API. The mobile actor API consist of two packages: *jmas.actor*, and *jmas.util.* Each package contains the classes needed inorder to write mobile actor programs. Both packages must be imported in every mobile actor program.

B.1 Elements of the *jmas.actor* API

The Mobile Actor API provides the methods to interface with actors within the JMAS environment. This API allows users to create actors (local or remote), replace actor behavior (local or remote), terminate an actor, and send a communication between actors. The API also provides methods which allow actors to get the available host from the CPU market. There are also methods which allow an actor to query the local machine, or a remote machine for the current load. Given these functions, actors can be written as intelligent agents; capable of making their own decisions on where to create remote actors, or the next location to continue processing.

The classes of the actor API are as follows:

- *Actor*
- *MobileActor*

Conceptually, an actor can be thought of as an object with the following data member, a *mail address*, and the following functional methods:*send()*,*create()*, and *become().* The semantics of these operations follow the standard actor definition of the primitive operations given in [Agh86]. Objects which extent from this class may only perform their methods on the local machine. These constructs provide useful mechanisms for writing concurrent/parallel programs on workstations or multiprocessor systems which may not have network access to remote resources.

A mobile actor has all the characteristics and functionality of basic actors with the extension of the following two functional methods: *becomeRemote()* and *createRemote().* The semantics of these operations follow those given in [BG98]. Objects which extent from this class may perform their methods on the local, as well as a remote machine. As illustrated in Figure A.1, we can represent actors and mobile actors using an Inheritance hierarchy. Below we give the interface of the actor API

for writing actor and mobile actor programs. These interfaces provide the only methods needed in order to develop actor/mobile actor applications. We also give a simple example of using constructs from each API.

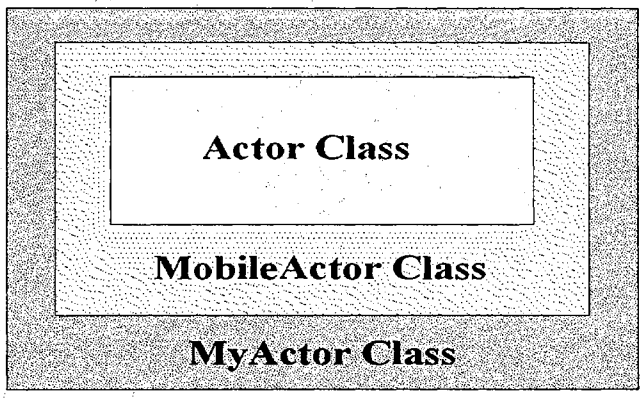


Figure B.1. Is-a Relationship of Actors Objects using Inheritance.

B.1.1 Actor Class

Class jmas.actor.Actor implements *Serializable*

Description: *Implements actor semantics on the local machine*

Data Members:

```
protected ActorAddress addr;
```

Constructors:

```
none
```

Methods:

```
public  ActorAddress getAddress();
-- returns the address of itself (implied)

public  void sink();
-- terminate or destroy actor

public  void send(ActorAddress  a, String  comm);
-- send comm to  Actor a

public  void send(ActorAddress a , String  comm, Object[] param);
-- send comm & param[] to Actor a
```

```
public  void send(ActorAddress a  String  comm, Object param);
-- send comm & param to Actor a

public  void send(String comm);
-- send comm to itself

public  ActorAddress create(String behv, Object[] acq)
-- create behv with acq[] on local machine
-- return identity

public  ActorAddress create(String behv, Object acq)
-- create behv with acq on local machine
-- return identity

public  ActorAddress create(String behv)
--  create behv on local machine
--  return identity

public  void become(String behv, Object[] acq)
--  become new behv with acq[] on local machine

public  void become(String behv, Object acq)
--  become new behv with acq on local machine

public  void become(String behv)
--  become new behv  on local machine
```

Example of Usage

In the following example, the current actor assumes the behavior HelloActor and receives a communication *hello*. Upon receipt of the communication, it executes the *hello* method. The current actor then creates a new actor W, which assumes the behavior WorldActor with acquaintance "Hello". The actor then terminates. The *sink()* operation is implied when the life of a thread is over. The new actor W, instantiates itself using the constructor that matches the type and number of acquaintances. Upon receipt of the communication (method="world", parameter="World!!") it executes the method *world* .

```
public class HelloActor extends Actor
{
    public void boot() { hello(); }

    public void hello()
    {
         ActorAddress  W = create("WorldActor", new String("Hello"));
         send(W,"world", new String("World!!"));
    }
}

public class WorldActor extends Actor
{
    String h;

    public WorldActor(String x) {  h  = x; }

    public void world(String w) { System.out.println(h + w); }
}
```

The above example demonstrates the use of the actor API to write an actor program. The output of the example is - Hello World!! More examples are shown in Appendix C.

B.1.2 MobileActor Class

Class **jmas.actor.MobileActor** implements *Serializable*

Description: *Implements mobile actor semantics on the local machine*

Data Members: **Constructors:**

none

Methods:

```
ActorAddress getAvailableHost();
--  returns the address of an available machine within the CPU market
    (Dynamic placement)

String getLocalHost();
-- returns the address of the local machine

public  int getLocalLoad()
-- returns the load on the local machine

public  int getRemoteLoad(ActorAddress loc)
-- returns the load on the remote machine at loc

public  ActorAddress createRemote(String behv, ActorAddress loc, Object[] acq)
--  create remote behv with acq[] on machine loc
--  return identity

public  ActorAddress createRemote(String behv, ActorAddress loc, Object acq)
--  create remote behv with acq on machine loc
--  return identity

public  ActorAddress createRemote(String behv, ActorAddress loc)
--  create remote behv on machine loc
--  return identity

public  ActorAddress createRemote(String behv, String  loc, Object[] acq)
--  create remote behv with acq[] on machine loc
--  return identity

public  ActorAddress createRemote(String behv, String loc, Object acq)
--  create remote behv with acq on machine loc
--  return identity

public  ActorAddress createRemote(String behv, String loc)
--  create remote behv on machine loc
--  return identity

public  void becomeRemote(String behv, ActorAddress loc ,Object[] acq)
--  become new behv with acq[] assume identity of loc

public  void becomeRemote(String behv, ActorAddress loc, Object acq)
--  become new behv with acq assume identity of loc

public  void becomeRemote(String behv, ActorAddress loc)
--  become new behv  assyme identity of loc
```

Example of Usage

Example: In the following example, the current actor assumes the behavior Hello World and receives a communication *hello*. Upon receipt of the communication, it executes the *hello* method. The current actor then creates a new actor W, on a remote machine, which assumes the behavior HelloWorld

with acquaintance "Hello". The actor then terminates. The *sink()* operation is implied when the life of a thread is over. The new actor W, instantiates itself using the constructor that matches the type and number of acquaintances. Upon receipt of the communication (method="world", parameter="World!!") it executes the method *world.*

```
public class HelloWorld extends MobileActor
{
    String h;

    public HelloWorld(String x) { h = x; }

    public void hello()
    {
          String acq = new String("Hello"));

          ActorAddress  W = createRemote("HelloWorld", getAvailHost(), acq);
          send(W,"world", new String("World!!"));
    }

    public void world(String w) { System.out.println(h + w); }
}
```

The above example demonstrates the use of the JE-aAPI to write a mobileactor program. The output of the example is - Hello World!! We use the method *getAvailHost()* to dynamically determine the remote location on which to create the mobile actor. A static placement strategy could also be used by explicitly specifying the hostname. Notice that we packaged the method *world* within the behavior HelloWorld. This gets rid of the extra network transmission needed to load the Behavior WorldActor over the network from the original location. When writing programs that involve several behaviors, take note that access to behaviors which are not on the local machine need to be loaded from a remote location causing additional communication overhead. More examples are shown in Appendix C.

B.2 Elements of the *jmas.util* API

B.2.1 The ActorAddress Class

The ActorAddress class contains the address or identity of an actor. When creating actors an identity is returned in the following format.

Mailbox@*location*

Class **jmas.util.ActorAddress** implements *Serializable*

Description: *contains the address of an actor*
Data Members:

```
private String mailbox;
private String location;
```

Constructors:

```
public ActorAddress()

public ActorAddress(String mailbox, String loc)
--  creates an actor address mailbox@loc

public ActorAddress(String id)
--  creates an actor address id = mailbox@location
```

Methods:

```
public String getMailbox()
--  return the mailbox of an actor address

public String getLocation()
--  return the location of an actor address

public String getIdentity()
--  return the full actor address

public void setLocation(String l)
--  sets the location = l of an actor address

public void setMailbox(String m)
--  sets the mailbox = m of an actor address

public void setAddress(String id)
--  sets the  actor address mailbox@location = id

public String toString()
--  converts actor address to a string

public boolean equals(Object)
--  comparison operator
```

APPENDIX C

EXAMPLE MOBILE ACTOR PROGRAMS

C.1 Hello World

The following example illustrates the use of the **boot()** method to start an actor program on the local machine (Line 9). Notice we imported two packages: *jmas.util*, and *jmas.actor*(Lines 4,5). These packages are needed in every actor/mobile actor program. Every actor program must extend from the **Actor** class (local computation) or the **MobileActor** class (distributed computation)(Line 7). All objects in JMAS must be serialized. Therefore, you must represent all basic data types using Java objects (Line 13). We create an actor x assuming the behavior cHelloActor, with a list of acquaintances acqs (Line 15). We send x a communication "hello" and a parameter – integer[59] (Line 17).

:::::::::::::: **cHelloWorld.java** ::::::::::::::

```
1 import  java.io.*;
2 import  java.lang.*;
3
4 import  jmas.util.*;
5 import  jmas.actor.*;
6
7 public class cHelloWorld extends Actor
8 {
9       public  void boot()
10       {
11           System.out.println("Starting [Concurrent]..... ");
12
13           Object[] acqs = new Object[] { new Integer(10), new Double(34.5) };
14
15            ActorAddress x = create("cHelloActor" , acqs);
16
17             send(x, "hello" , new Integer(59));
18       }
19 }
```

Upon creation of an actor assuming the behavior cHelloActor, its constructor is executed in order to pass the acquaintances as parameters to the object(Line 12). After receiving the communication ["hello" , Integer(59)], it executes the method *hello* with the integer parameter 59(Line 18). This method in turn creates two additional actors with behaviors: cHelloActor, and cWorldActor with an acquaintance x (Line 22,24). It then sends two communications:

1. ["*world*"*Integer*(25)] $\Rightarrow$ to the cHelloActor (Line 26)

2. ["*hello*"] $\Rightarrow$ to the cWorldActor (Line 28)

When the cHelloActor receives the communication "world" Integer (25), it executes the world

method of the class cHelloActor (Line 32). The "world" method creates an actor with a behavior cWorldActor and an acquaintance x; then sends it a communication ["hello"](Line 38,39).

:::::::::::::: **cHelloActor.java** ::::::::::::::

```
1 import  java.io.*;
2 import  java.lang.*;
3
4 import  jmas.util.*;
5 import  jmas.actor.*;
6
7 public class cHelloActor extends Actor
8 {
9       int z;
10       double w;
11
12       public cHelloActor(Integer x, Double y)  // pass two acquaintances
13       {
14            z = x.intValue();
15            w = y.doubleValue();
16       }
17
18       public void hello(Integer x)
19       {
20           System.out.println("Hello [Param] -- " + x.intValue() +
21                " [ACQs] --- " + z + " & " + w);
22           ActorAddress w = create("cHelloActor");
23
24           ActorAddress h = create("cWorldActor", x);
25
26           send(w,"world", new Integer(25));
27
28           send(h,"hello");
29
30          // since there are no operations then sink
31       }
32       public void world(Integer x)
33       {
34           System.out.println("WORLD! -- [Parms] " + x.intValue());
35
36         //  send a message to the replacement behv
37
38           ActorAddress h = create("cWorldActor", x);
39           send(h, "hello");
40
41           // since there are no operations then sink
42       }
43 }
```

Upon creation of an actor assuming the behavior cWorldActor, its constructor is executed in order to pass the acquaintances as parameters to the object(Line 13). After receiving the communication ["hello"], it executes the method "hello()"(Line 18).

:::::::::::::: **cWorldActor.java** ::::::::::::::

```
1 import  java.io.*;
2 import  java.lang.*;
3
4 import  jmas.util.*;
5 import  jmas.actor.*;
6
7 public class cWorldActor extends Actor
```

```
8 {
9       // Acquaintances
10
11       public int z;
12
13       public cWorldActor(Integer x)
14       {
15           z = x.intValue();
16       }
17
18       public void hello()
19       {
20           System.out.println("WORLD!" + " [acq] " + z);
21       }
22 }
```

C.2 TravelTime

The following example illustrates how to write a mobile actor program (Line 7). We ran our program from machine a.cs.okstate.edu. The boot actor creates a remote actor x on machine *z.cs.okstate.edu*; assuming the behavior TravelTime (Line 19). It sends an initial communication "Timeit" along with a parameter "the @ddress of itself" (Line 21). The current actor becomes a new actor assuming the behavior "TravelTime" with an acquaintance a Long integer (Line 22). Upon receipt of the communication ["Timeit" and c= "the @ of itself"], actor x executes the method "Timeit" (Line 25). The actor then sends a communication to the actor c , with the communication ["TotalTime" and an Integer d] (Line 31). The initial actor who replaced its behavior receives the communication and executes the method "TotalTime" (Line 34).

:::::::::::::::::: TracelTime.java ::::::::::::::::::

```
1 import  java.io.*;
2 import  java.lang.*;
3
4 import  jmas.util.*;
5 import  jmas.actor.*;
6
7 public class TravelTime extends MobileActor
8 {
9       long start = 0;
10
11       public TravelTime(Long s) { start = s.longValue(); }
12
13       public void boot()
14       {
15           Long t = new Long(System.currentTimeMillis());
16           System.out.println("Starting [Concurrent2]..... ");
17           System.out.println("\n\n....PEEK ");
18
19           ActorAddress x = createRemote("TravelTime","z.cs.okstate.edu");
20
21           send(x, "Timeit" , getAddress());
22                become("TravelTime",t);
23       }
24
25       public  void Timeit(ActorAddress c)
26       {
27                Long t = new Long(System.currentTimeMillis());
28
29           System.out.println("\n\n....A ");
30
31           send(c,"TotalTime", t);
32       }
33
34        public  void TotalTime(Long l)
35       {
36           System.out.println("\n\n....BOO! ");
37
38            long i = l.longValue();
39
40           System.out.println("Actor A: behv[boot] --> behv[Timeit]\n");
41           System.out.println("\tIt took: " + (i - start)/1000.0 + " msec");
42
```

```
43            System.out.println("Actor A: behv[Timeit] --> behv[TravelTime]\n");
44                 long c = System.currentTimeMillis();
45            System.out.println("\tIt took: " + (c - i)/1000.0 + " msec");
46
47            System.out.println("Total Exec Time took: " + (c - start)/1000.0 + " msec");
48       }
49 }
```

C.3 Parallel Sum

The following example illustrates how mobile actors can be used to decompose a computation to sum a set of integers in parallel. Our decomposition process organizes actors as a hierarchy of $2^k - 1$ nodes; where k is the number of levels in the hierarchy. Each actor with the exception of the root actor, is responsible for creating two additional actors to handle the subcomputations. We create a remote actor x using dynamic processor placement by issuing the function *getAvailableHost*(). The actor x assumes the behavior *Sum*, with the acquaintances p: the address to return the sum, the number of inputs to return, and the granularity of the computation (Line 37). We send x a communication "decompose" and a parameter – the list of intergers to sum *list* (Line 39). The current actor then changes its state to itself, in order to wait for a communication to print the result.

:::::::::::::::::::: **PrintSum.java** ::::::::::::::::::::

```
1 import  java.io.*;
2 import  java.util.*;
3 import  java.lang.*;
4
5 import  jmas.util.*;
6 import  jmas.actor.*;
7
8 public class PrintSum extends MobileActor
9 {
10       Long time;
11
12       final int SIZE = 10000;
13       final int procs = 2;
14
15       public PrintSum() { }
16
17       public PrintSum(Long t) { time = t; }
18
19       public void boot()
20       {
22           System.out.println("Starting [Distributed Sum]..... ");
23
24           Long[] list = new Long[SIZE];
25
26           Random id = new Random(9000);
27           long v;
28
29           System.out.print("The Sum of " + SIZE + " numbers is:  ");
30
31           for(int i = 0;i < SIZE; i++)
32           {
33              v = id.nextLong() % 200;
34              list[i] = new Long(v);
35           }
36           Object[] p = new Object[] {getAddress(), new Integer(1),
                 new Integer(SIZE/procs)});
37           ActorAddress x = createRemote("Sum" , getAvailableHost(), p);
38
39           send(x, "decompose" ,  (Object) list);
40
```

```
41              become("PrintSum", new Long(System.currentTimeMillis()));
42          }
43
45          public void print(Long sum)
46          {
47              System.out.println(sum.longValue());
48
49              float ftime = System.currentTimeMillis() - time.longValue();
50              System.out.println("\n Processing time: " + ftime/1000.0 + " sec");
51              System.out.println("\n Load time: " + getLoadTime() + " sec");
52          }
53 }
```

A mobile actor assuming the behavior *Sum* that receives the communication "decompose", executes the method *decompose* in the class *Sum*. If the list has reached a size that is equal to the granularity of the computation then the summation process begins and the result is returned to the calling actor (Lines 40-43). If the list has not reached a size equal to the granularity, the list is decomposed again into two equal parts (Lines 50 and 51) and sent to two newly created actors. Actors are dynamically placed within the system using the function *getAvailHost*(). After an actor decomposes the data set, it changes its state to itself in order to wait for a communication (*MultiInputAdder*) to receive the multiple inputs (Lines 49). Because actors perform operations in response to one communication, we build on the primitive data types in order to implement higher level abstractions such as multiple input actors (Lines 65-84). In response to a communication "MultiInputAdder", an actor adds in the result to the partial sum. If the number of communications equal the number of actors created by the current actor during decomposition, then the sum is returned back to the calling actor (Line 78 or 80). The calling actor could be the root (Line 80), or an intermediate actor for processing the partial sum (Line 78). If the number of communications does not equal the number of actors created by the current actor during decomposition, the current actor changes its state to itself in order to wait for the next input (Line 83).

:::::::::::::::::::::::: **Sum.java** ::::::::::::::::::::::::

```
1 import  java.io.*;
2 import  java.lang.*;
3
4 import  jmas.util.*;
5 import  jmas.actor.*;
6
7 public class Sum extends MobileActor
8 {
9       ActorAddress cust = null;
10       Integer inputs = null;
11       Integer outputs = null;
```

```
12      Long sum = null;
13      Integer granularity;
14
15      public Sum() { }
16
17      public Sum(ActorAddress c, Integer o, Integer g)
18      {
19          cust = c;
20          outputs = o;
21          granularity = g;
22      }
23
24      public Sum(ActorAddress c, Long s, Integer in, Integer o)
25      {
26          cust = c;
27          inputs = in;
28          outputs = o;
29          sum = s;
30      }
31
32      public  void decompose(Long[] list)
33      {
35          long psum = 0;
36
38          if(list.length == granularity.intValue())
39          {
40              for(int i = 0;i < list.length;i++)
41                  psum += list[i].longValue();
42
43              send(cust,"MultiInputAdder", new Long(psum));
44          }
45          else
46          {
47              Long[] l1, l2;
48
49              become("Sum", new Object[] { cust, new Long(0),
                        new Integer(2), outputs} );
50              ActorAddress p1 = createRemote("Sum", getAvailableHost(),
                        new Object [] {getAddress(), new Integer(2),granularity });
51              ActorAddress p2 = createRemote("Sum", getAvailableHost(),
                        new Object [] {getAddress(), new Integer(2),granularity });
52
53              l1 = new Long[list.length/2];
54              l2 = new Long[list.length/2];
55
56              System.arraycopy(list,0,l1,0,list.length/2);
57              System.arraycopy(list,list.length/2,l2,0,list.length/2);
58
59              send(p1,"decompose", (Object) l1);
60              send(p2,"decompose", (Object) l2);
61          }
62      }
63
65      public  void MultiInputAdder(Long val)
66      {
67          //  send a message to the replacement behv
68          long psum = sum.longValue();
69          int in = inputs.intValue();
70          int out = outputs.intValue();
71
72          --in;
73          psum += val.intValue();
74
75          if(in == 0)
76          {
77                if(out == 2)
78                    send(cust,"MultiInputAdder",new Long(psum));
79                else
80                    send(cust,"print",new Long(psum));
81          }
82          else
83                become("Sum",new Object[] {cust,new Long(psum),
                          new Integer(in), outputs});
84      }
85  }
```

C.4 Parallel Quicksort

The following example illustrates how mobile actors can be used to decompose a computation to sort a set of integers in parallel using the QuickSort Algorithm (Line 89-149). Our decomposition process organizes actors as a hierarchy of $2^k - 1$ nodes; where k is the number of levels in the hierarchy. Each actor with the exception of the root actor, is responsible for creating two additional actors to handle the subcomputations. We create a remote actor x using dynamic processor placement by issuing the function *getAvailableHost*(). The actor x assumes the behavior *Sort*, with the acquaintances: an address to return the final sorted list, and the number of inputs to return (Line 35 and 36). We send x a communication "decompose" and a parameter – the list of intergers to sort *list* (Line 37). The current actor then changes its state to itself, in order to wait for a communication to print the result.

:::::::::::::::::::::::: **PrintSort.java** ::::::::::::::::::::::::

```
1 import  java.io.*;
2 import  java.util.*;
3 import  java.lang.*;
4
5 import  jmas.util.*;
6 import  jmas.actor.*;
7
8 public class PrintSort extends MobileActor
9 {
10       Long time;
11
12       final int SIZE = 10000;
13
14
15
16       public PrintSort() { }
17
18       public PrintSort(Long t) { time = t; }
19
20       public void boot()
21       {
22           System.out.println("Starting [Distributed Sort]..... ");
23
24           Long[] list = new Long[SIZE];
25
26           Random id = new Random(9000);
27           long v;
28
29           for(int i = 0;i < SIZE; i++)
30           {
31              v = id.nextLong();
32              list[i] = new Long(v);
33           }
34
35           ActorAddress x = createRemote("Sort" , getAvailableHost(),
36           new Object[] {getAddress(), new Integer(1)});
37           send(x, "decompose" ,  (Object) list);
38
39           become("PrintSort", new Long(System.currentTimeMillis()));
40       }
```

```
41
42
43        public void print(Long[] flist)
44        {
45            System.out.println("\n\nThe Sorted List size " + SIZE + ": ");
46
47
48            float ftime = System.currentTimeMillis() - time.longValue();
49
50            for(int i = 0;i < flist.length;i++)
51                System.out.println(" , " + flist[i]);
52            System.out.println("\n\n Processing time: " + ftime/1000.0 + " sec");
53            System.out.println("\n Load time: " + getLoadTime() + " sec");
54        }
55 }
```

A mobile actor assuming the behavior *Sort* that receives the communication "decompose", executes the method *decompose* in the class *Sort*. If the list has reached the size 500, then the sorting process begins and the result is returned to the calling actor (Lines 40-43). If the list has not reached the size 500, the list is decomposed again into two equal parts (Lines 50 and 51) and sent to two newly created actors. Actors are dynamically placed within the system using the function *getAvailHost*(). After an actor decomposes the data set, it changes its state to itself in order to wait for a communication (*MultiInputSort*) to receive the multiple inputs (Lines 49). Because actors perform operations in response to one communication, we build on the primitive data types in order to implement higher level abstractions such as multiple input actors (Lines 65-84). In response to a communication "MultiInputSort", an actor merges the result of two partial lists that have been sorted. If the number of communications equal the number of actors created by the current actor during decomposition, then the sorted list is returned back to the calling actor (Lines 81 and 83). The calling actor could be the root (Line 83) , or an intermidiate actor for processing the partial sum (Line 81). If the number of communications does not equal the number of actors created by the current actor during decomposition, the current actor changes its state to itself in order to wait for the next input (Line 86).

:::::::::::::::::::::::: **Sort.java** ::::::::::::::::::::::::

```
1 import  java.io.*;
2 import  java.lang.*;
3
4 import  jmas.util.*;
5 import  jmas.actor.*;
6
7 public class Sort extends MobileActor
8 {
9       ActorAddress cust = null;
10       Integer inputs = null;
```

```
11      Integer outputs = null;
12      Long[] pl = null;
13
14      public Sort() { }
15
16      public Sort(ActorAddress c, Integer o)
17      {
18          cust = c;
19          outputs = o;
20      }
21
22      public Sort(ActorAddress c, Long[] l, Integer in, Integer o)
23      {
24          cust = c;
25          inputs = in;
26          outputs = o;
27          pl = l;
28      }
29
30      public Sort(ActorAddress c, Integer in, Integer o)
31      {
32          cust = c;
33          inputs = in;
34          outputs = o;
35      }
36
37      public  void decompose(Long[] list)
38      {
39
40          if(list.length == 500)
41          {
42              QuickSort(list, 0, list.length - 1);
43              send(cust,"MultiInputSort", (Object) list);
44          }
45          else
46          {
47              Long[] l1, l2;
48
49              become("Sort", new Object[] { cust, new Integer(2), outputs} );
50              ActorAddress p1 = createRemote("Sort", getAvailableHost(),
                        new Object [] {getAddress(), new Integer(2) });
51              ActorAddress p2 = createRemote("Sort", getAvailableHost(),
                        new Object [] {getAddress(), new Integer(2) });
52
53              l1 = new Long[list.length/2];
54              l2 = new Long[list.length/2];
55
56              System.arraycopy(list,0,l1,0,list.length/2);
57              System.arraycopy(list,list.length/2,l2,0,list.length/2);
58
59              send(p1,"decompose", (Object) l1);
60              send(p2,"decompose", (Object) l2);
61          }
62      }
63
64
65      public  void MultiInputSort(Long[] plist)
66      {
67          //  send a message to the replacement behv
68          int in = inputs.intValue();
69          int out = outputs.intValue();
70
71          --in;
72
73          if(in == 0)
74          {
75                Long[] list = new Long[pl.length + plist.length];
76                System.arraycopy(pl,0,list,0,pl.length);
77                System.arraycopy(plist,0,list,pl.length,plist.length);
78                QuickSort(list, 0, list.length - 1);
79
80                if(out == 2)
81                    send(cust,"MultiInputSort", (Object)list);
82                else
83                    send(cust,"print",(Object)list);
```

```
84            }
85            else
86                  become("Sort",new Object[] {cust,(Object) plist,
                         new Integer(in), outputs});
87       }
88
89    public void QuickSort(Long a[], int lo0, int hi0)
90    {
91       int lo = lo0;
92       int hi = hi0;
93       Long mid;
94
95       if ( hi0 > lo0)
96       {
98          /* Arbitrarily establishing partition element as the midpoint of
99           * the array.
100           */
101          mid = a[ ( lo0 + hi0 ) / 2 ];
102
103          // loop through the array until indices cross
104          while( lo <= hi )
105          {
106             /* find the first element that is greater than or equal to
107              * the partition element starting from the left Index.
108              */
109              while( ( lo < hi0 ) && ( a[lo].longValue() < mid.longValue() ))
110                  ++lo;
111
112             /* find an element that is smaller than or equal to
113              * the partition element starting from the right Index.
114              */
115              while( ( hi > lo0 )  && ( a[hi].longValue() > mid.longValue() ))
116                  --hi;
117
118             // if the indexes have not crossed, swap
119             if( lo <= hi )
120             {
121                swap(a, lo, hi);
122                ++lo;
123                --hi;
124             }
125          }
126
127          /* If the right index has not reached the left side of array
128           * must now sort the left partition.
129           */
130          if( lo0 < hi )
131             QuickSort( a, lo0, hi );
132
133          /* If the left index has not reached the right side of array
134           * must now sort the right partition.
135           */
136          if( lo < hi0 )
137             QuickSort( a, lo, hi0 );
138
139       }
140    }
141
142    private void swap(Long a[], int i, int j)
143    {
144       Long T;
145       T = a[i];
146       a[i] = a[j];
147       a[j] = T;
148
149    }
150
151 }
```

C.5 Round Robin Migration through Market

The following example illustrates how mobile actors can be used to migrate through a network using an itinerary list. The itinerary list is stored in the array *itin* (Lines 10-30). When executing the following program the current actor migrates to the ith machine in the itinerary list using *createRemote*() and static placement. This continues until the all machines in the list have been visited.

:::::::::::::::::::::::: **roundrobin.java** ::::::::::::::::::::::::

```
1 import java.io.*;
2 import java.util.*;
3
4 import jmas.util.*;
5 import jmas.actor.*;
6
7 public class roundrobin extends MobileActor
8 {
9        Integer machine = null;
10        String[] itin =   new String[] {"a.cs.okstate.edu",
11                                        "esws01.wslab.okstate.edu",
12                                        "esws02.wslab.okstate.edu",
13                                        "esws03.wslab.okstate.edu",
14                                        "esws04.wslab.okstate.edu",
15                                        "esws05.wslab.okstate.edu",
16                                        "esws06.wslab.okstate.edu",
17                                        "esws07.wslab.okstate.edu",
18                                        "esws08.wslab.okstate.edu",
19                                        "esws09.wslab.okstate.edu",
20                                        "esws10.wslab.okstate.edu",
21                                        "esws12.wslab.okstate.edu",
22                                        "esws13.wslab.okstate.edu",
23                                        "esws14.wslab.okstate.edu",
24                                        "esws15.wslab.okstate.edu",
25                                        "eslabsvr.wslab.okstate.edu",
26                                        "esws09.wslab.okstate.edu",
27                                        "a.cs.okstate.edu",
28                                        "z.cs.okstate.edu",
29                                        "esws06.wslab.okstate.edu",
30                                        "esws13.wslab.okstate.edu", };
31
32        public roundrobin() { machine = new Integer(0); }
33
34        public roundrobin(Integer i)
35        {
36           machine = i;
37        }
38
39        public void boot()
40        {
41                if(machine.intValue() < itin.length)
42                {
43             System.out.println("Hello I'm at machine: " + itin[machine.intValue()]);
44             int num = machine.intValue() + 1;
45             ActorAddress addr = createRemote("roundrobin",itin[num], new Integer(num));
46             send(addr,"boot");
37                }
48        }
49 }
```

C.6 Traveling Salesman Problem

The following example illustrates how mobile actors are used to compute a solution to the Traveling Salesman Problem in parallel. Our solution is based on a brute force exhaustive search method. Given a problem size for N cities and p machines, we decompose all $(N-1)!$ tours such that each machine performs a exhaustive search on $(N-1)!/p$ tours. Machines are organized using a master-slave design. Given n machines the master distributes the subcomputations to the $n-1$ slaves and waits for the best possible tour computed from each slave. In addition, the master also computes the best possible tour from a given set of tours. The program below decomposes the a problem size for N cities across $2*N$ processes. The decomposition process is done on Lines (55-71 *TSP.java*). Remote actor creation is performed using the function *createRemote*(); actor placement is dynamic (Lines 58 and 68 *TSP.java*). After program decomposition, the current actor changes its state to itself in order to wait for the result (Line 74 *TSP.java*). Upon receipt of a communication "result", the method *result* in the class "TSP" is executed. Results are collected as multiple inputs and use a strategy similar to the Parallel Sort/Sum algorithms (Section C.4 and C.5).

:::::::::::::::::::::::: **TSP.java** ::::::::::::::::::::::::

```
1 import java.io.*;
2 import java.util.*;
3
4 import jmas.util.*;
5 import jmas.actor.*;
6
7 public class TSP extends MobileActor
8 {
9         final int N = 13;
10         final int n = N-1;
11         final int procs = 2*n;
12
13         Long time;
14         int in;
15         Permutation best = new Permutation(N-1);
16         double bd;
17         double load;
18
19         public TSP() { }
20
21         public TSP(Long t, Integer r, String bst, Double b, Double l)
22         {
23              time = t;
24              in = r.intValue();
25              best.set(bst);
26              bd = b.doubleValue();
27              load =l.doubleValue();
28         }
29
30         public TSP(Long t)
31         {
32              time = t;
33              in = procs;
```

```
34              bd = Double.MAX_VALUE;
35              load = 0.0;
36          }
37
38          public void boot()
39          {
40              TSP_Problem prob = new TSP_Problem();
41              Double[][] distancevector = prob.randomProb(N);
42              ActorAddress addr = new ActorAddress();
43              Object[] param;
44
45
46              int tours = fact(N);
47
48              if(procs <= tours)
49              {
50                  System.out.println("TSP 2n proc!.....");
51                  Permutation p = new Permutation(n);
52                  Permutation fin = new Permutation(n);
53
54                  fin.reset();
55                  for(int i = 0;i < n;i++)
56                  {
57                      param = new Object[] {distancevector, fin.toString()};
58                      addr = createRemote("TSPSolver",getAvailableHost(), param);
59                      int y = n/2;
60                      if(n/2 == i)
61                          y++;
62
63                      fin.reset(i,y);
64                      param = new Object[] {fin.toString(), getAddress()};
65                      send(addr,"improve", param);
66
67                      param = new Object[] {distancevector, fin.toString()};
68                      addr = createRemote("TSPSolver",getAvailableHost(), param);
69                      fin.reset(i+1);
70                      param = new Object[] {fin.toString(), getAddress()};
71                      send(addr,"improve", param);
72                  }
73                  time = new Long(System.currentTimeMillis());
74                  become("TSP", time);
75              }
76              else
77                  System.out.println("<nprocs> <= <N-1>!");
78          }
79
80          public int fact(int n)   // only include n-1! tours
81          {
82              int v = 1;
83              for(int i=1;i<n;i++)
84                  v *= i;
85
86              return v;
87          }
88
89          public void result(String perm, Double bestDistance, Double ld)
90          {
91              //check the best dist, then become
92              double bestd = bestDistance.doubleValue();
93
94              if(bestd < bd)
95              {
96                  best.set(perm);
97                  bd = bestd;
98              }
99
100             load += ld.doubleValue();
101
102             if(--in > 0)
103                 become("TSP", new Object[] { time, new Integer(in),
                            best.toString(), new Double(bd), new Double(load) });
104             else
105             {
106                 long f = System.currentTimeMillis();
107                 System.out.println("The Optimal solution is <" +
                        (N-1) + "> " + best.toString());
```

```
108                     System.out.println("The Distance  is " + bd);
109                     System.out.println("Elapse Time: " +
                            (float)((f - time.longValue())/1000.0) + " secs");
110                     System.out.println("%Load Time: " +
                            (load + getLoadTime()) + " secs");
111
112                 }
113         }
114 }
```

Each subcomputation is an actor *TSPSolver* (*TSPSolver.java*). Upon receipt of a communication "improve", the actor executes an exhaustive search process on a subset of the total possible tours (Lines 128-142). No two actors perform a search on the same subset. After computing the best possible tours, this value is returned to the master (Line 141 *TSPSolver.java*).

:::::::::::::::::::::::: **TSPSolver.java** ::::::::::::::::::::::::

```
1 // Solver.java
2 import java.util.Vector;
3 import jmas.util.*;
4 import jmas.actor.*;
5
6 /**
7  * Traveling Salesman Problem:  a brute-force "exhaustive search".
8  * Solutions are to be represented as Permutations that indicate
9  * the order in which to visit all but one of the cities in the
10  * original list.  The last city is kept fixed.
11  */
12
13 public class TSPSolver extends MobileActor{
14
15         public TSPSolver(Double[][] d, String begin)
16         {
17                 distances = d;
18                 size = distances.length;
19                 startSolver(begin);
20         }
21
22         /**
23          * Create two permutations to encode "tours" of the cities.  That
24          * is, each will indicate an order in which the cities are to be
25          * visited.  One is used to hold the best tour found so far, and
26          * the other is used as a "temporary" permutation for trying out
27          * new tours.  The permutations are one shorter than the list of
28          * cities, since we can keep one city fixed to simplify the search.
29          */
30         public void startSolver(String s) {
31                 tmp = new Permutation(size-1);
32                 best = new Permutation(size-1);
33                 resetTour(s);
34         }
35
36         /**
37          * Return the distance associated with the best tour so far.
38          */
39         public  double distance() {
40                 return bestDistance;
41         }
42
43         /**
44          * Return the number of tours (or "configurations") tested so far.
45          */
46         public  long configs() {
47                 return count;
48         }
49
50         /**
51          * Go back to the initial random tour by resetting the permutations
```

```
52          * back to the identity.  Also reset things like the distance.
53          */
54         public void resetTour(String x) {
55                 count = 0;
56                 done = false;
57                 best.set(x);
58                 tmp.set(x);
59                 bestDistance = calcDistance(best);
60         }
61
62         /**
63          * Try out a number of new tours (i.e. the number specified by
64          * "steps").  This is done in such a way that we will eventually
65          * try every possible tour with no duplication.  If a better tour
66          * is found, save it along with its length and return true.
67          * Otherwise, return false.
68          */
69         public boolean exhaustiveImprove(Permutation e) {
70                 double newDist;
71                 boolean retval = false;
72
73                 while(retval == false && done == false) {
74                         newDist = calcDistance(tmp);
75                         if(newDist < bestDistance) {
76                                 retval = true;
77                                 bestDistance = newDist;
78                                 best.set(tmp);
79                         }
80                         ++count;
81                         if(tmp.next() == false || tmp.equals(e)) {
82                                 done = true;
83                                 break;
84                         }
85                 }
86                 return retval;
87         }
88
89         /**
90          * Return true if the current tour is known to be optimal (e.g. all
91          * possible tours have been checked).
92          */
93         public String optimality() {
94                 if(done)
95                         return "Optimal";
96                 return "In Progress";
97         }
98
99         /**
100          * Return the permutation encoding the best tour discovered so far.
101          * This permutation applies to all but the last city, which is
102          * fixed.
103          */
104         public Permutation currentTour() {
105                 return best;
106         }
107
108         // The private Methods:
109
110         /*
111          * Calculates the total distance for any particular permutation.  This
112          * uses table lookup to get the distances between two points.
113          */
114         private double calcDistance(Permutation p) {
115                 int p1, p2;
116                 double accum = 0;
117
118                 p1 = p2 = size - 1;
119                 for(int i = 0; i < size - 1; ++i) {
120                         p2 = p.index(i);
121                         accum += distances[p1][p2].doubleValue();
122                         p1 = p2;
123                 }
124                 accum += distances[size - 1][p2].doubleValue();
125                 return accum;
126         }
127
128         public void  improve(String end, ActorAddress cust) {
```

```
129                 boolean improved = true;
130                 Permutation e = new Permutation(size-1);
131                 e.set(end);
132
133                 while(improved)
134                 {
135                     improved = exhaustiveImprove(e);
136                 }
137
138                 Double ltime = new Double(getLoadTime());
139
140                 Object[] param = new Object[] { best.toString(),
                        new Double(distance()), ltime};
141                 send(cust,"result",param);
142         }
143
144         // The private data:
145
146         /*
147          * All distances between all points are pre-calculated and stored in
148          * this array.  Math.sqrt() only gets called N^2/2 times total for any
149          * particular problem.
150          */
151         private Double distances[][];
152
153         private Permutation tmp;
154         private Permutation best;
155         private int size;
156         private int count;
157         private boolean done;
158         private double bestDistance;
159 }
160
```

::::::::::::::::::::::::: **TSPProblem.java** :::::::::::::::::::::::::

```
1 // TSP_Problem.java
2
3 import java.util.Vector;
4 import java.util.Random;
5 import java.io.*;
6
7 /**
8  * This class generates problem instances for the Traveling Salesman
9  * Problem.  Each call to "randomProb" will return a
10  * Vector of 2D points in the unit square [0,1]x[0,1] which are
11  * guaranteed to be separated by some small distance (i.e. the points
12  * will not be too clumped up).
13  */
14
15 class TSP_Problem implements Serializable{
16         private double SepDist = 0.04;  // Minimum separation of points.
17         private Random rng;
18         private Vector points;
19         private int seed = 75825;
20         private int size;
21
22         public TSP_Problem() {
23                 points = new Vector();  // This will hold new problem instances.
24                 rng = new Random(seed); // This is a random number generator.
25         }
26
27         // Pick n random points in [0,1]x[0,1] that are separated by
28         // a small distance (so that we don't get clumps of points).
29         public Double[][] randomProb(int n) {
30                 size = n;
31                 Point2D Q = new Point2D();   // This is a temporary variable.
32                 points.removeAllElements();  // Blow away the old vector.
33                 for(int npts = 0; npts < n;) {
34                         boolean too_close = false;
35                         Q.set(rng.nextFloat(), rng.nextFloat());
36                         for(int j = 0; j < npts; j++) {
37                                 Point2D P = (Point2D)points.elementAt(j);
38                                 if(P.dist(Q) < SepDist) {
39                                         too_close = true;
40                                         break;
```

```
41                              }
42                      }
43                      if(!too_close) {
44                              // Create a new 2D point containing the
45                              // coordinates of Q, and add it to the growing
46                              // vector of points.  This, in effect, adds a
47                              // pointer to Q, so it's important to create a
48                              // new Point2D object to hold the coordinates.
49                              points.addElement(new Point2D(Q));
50                              npts++;
51                      }
52              }
53              return initDistances();
54      }
55
56      /*
57       * Initialize the distance table.
58       */
59      private Double[][] initDistances() {
60              Double[][] distances = new Double[size][size];
61              Point2D px;
62              Point2D py;
63
64              for(int y = 0; y < size; ++y) {
65                      py = (Point2D)points.elementAt(y);
66                      // Store the distances twice (square matrix) only
67                      // calculate them once.
68                      for(int x = y; x < size; ++x) {
69                              px = (Point2D)points.elementAt(x);
70                              distances[y][x] = new Double(px.dist(py));
71                              distances[x][y] = new Double(px.dist(py));
72                      }
73              }
74              return distances;
75      }
76
77 }
```

:::::::::::::::::::::::: Permutation.java ::::::::::::::::::::::::

```
1 // Permutation.java
2
3 import java.io.*;
4 import java.util.*;
5
6 /**
7  * This class encapsulates integer permutations of any size.  Each
8  * permutation is of the form (0,1,2,...,n-1), where n is the "size" of
9  * permutation.  The method "next" steps the permutation through all
10  * distinct configurations, hitting each one exactly once.  This is a
11  * convenient method for generating all possible permutations of n
12  * integers.
13  */
14
15 public class Permutation implements Serializable{
16         private int p[];
17         private int n;
18         private StringBuffer buff;
19
20         /**
21          * This creates a permutation of n integers, 0, 1, ... , n-1.
22          */
23         public Permutation(int size) {
24                 n = size;
25                 p = new int[n];
26                 buff = new StringBuffer();
27                 reset();
28         }
29
30         /**
31          * Returns the number of elements in the permutation.  A value of n
32          * means that the permutation consists of the integers
33          * (0,1,...,n-1).
34          */
35         public int size() {
36                 return n;
```

```
37          }
38
39          /**
40           * Returns the integer in the last position of the current
41           * permutation.
42           */
43          public int lastIndex()  {
44                  return p[n - 1];
45          }
46
47          /**
48           * Returns the integer in the i'th position of the current
49           * permutation.
50           */
51          public int index(int i) {
52                  return p[i];
53          }
54
55          /**
56           * Creates the "first" permutation, which is (0,1,...,n-1).
57           */
58          public void reset() {
59                  for(int i = 0; i < n; i++)
60                          p[i] = i;
61          }
62
63          /**
64           * Creates the first "x" permutation, which is (x,1,...,n-1).
65           */
66          public void reset(int x) {
67                  int j = 1;
68                  p[0] = x;
69                  for(int i = 0; j < n; i++, j++)
70                  {
71                          if(i == x)
72                                  p[j] = ++i;
73                          else
74                                  p[j] = i;
75                  }
76          }
77
78          public void reset(int x, int y) {
79                  int j = 2;
80
81                  p[0] = x;
82                  if(y == x)
83                      y++;
84                  p[1] = y;
85
86                  for(int i = 0; j < n; i++, j++)
87                  {
88                          if(i == x || i == y)
89                          {
90                                  i++;
91                                  if(i == x || i == y)
92                                      p[j] = ++i;
93                                  else
94                                      p[j] = i;
95                          }
96                          else
97                                  p[j] = i;
98                  }
99          }
100
101          /**
102           * set the current permutation based on the string
103           */
104          public void set(String s) {
105                  StringTokenizer tok = new StringTokenizer(s," ");
106                  int size = tok.countTokens() - 2;
107
108                  if(size == n)
109                  {
110                          tok.nextToken();
111                          for(int t=0;t < n;t++)
112                                  p[t] = Integer.parseInt(tok.nextToken());
```

```
113             }
114             else
115                     reset();
116     }
118     /**
119      * Copy another permutation into this one, even if it is a
120      * different size.  If the size if different, reallocate space for
121      * the integer array.
122      */
123     public void set(Permutation Q) {
124             if(n != Q.size()) {
125                     n = Q.size();
126                     p = new int[n];
127             }
128             for(int i = 0; i < n; i++)
129                     p[i] = Q.index(i);  // Copy the permutation Q.
130     }
131
132     public int[] getArray() {
133             return p;
134     }
135
136     public boolean equals(Permutation t)
137     {
138            boolean eq = true;
139
140            if(n == t.size())
141            {
142                int[] tmp = t.getArray();
143                for(int i = 0;i < n && eq;i++)
144                         if(p[i] != tmp[i])
145                                 eq = false;
146            }
147            else
148                eq = false;
149
150            return eq;
151     }
152
153     public boolean next()
154     {
155             int i = n - 1;
156             boolean atend = false;
157
158             while (p[i-1] >= p[i])
159             {
160                     i = i-1;
161                     if(i <= 0)
162                     {
163                         atend = true;
164                         break;
165                     }
166             }
167
168             if(atend)
169                     return false;
170
171             int j = n;
172
173             while (p[j-1] <= p[i-1])
174                     j = j-1;
175
176             swap(i-1, j-1); // swap values at positions (i-1) and (j-1)
177
178             i++; j = n;
179             while (i < j)
180             {
181                     swap(i-1, j-1);
182                     i++;
183                     j--;
184             }
185
186             return true;
187     }
189     /**
190      * This method creates a representation of the permutation as a
191      * string.  This is useful for demos and for debugging, but is not
```

```
192          * normally needed.
193          */
194         public String string() {
195                 buff.setLength(0); // Build up the string by appending.
196                 buff.append("( ");
197                 for(int i = 0; i < n; i++) {
198                         buff.append(p[i]);
199                         buff.append(' ');
200                 }
201                 buff.append(")");
202                 return buff.toString();
203         }
205         /**
206          * This method creates a representation of the permutation as a string.
207          * This is useful for demos and for debugging, but is not normally
208          * needed.  (It is also useful for "saving" a tour, since it encodes
209          * all the relevant information in a printable string.)
210          */
211         public String toString() {
212                 buff.setLength(0); // Build up the string by appending.
213                 buff.append( "( " );
214                 for(int i = 0; i < n; i++) {
215                         buff.append(p[i]);
216                         buff.append(' ');
217                 }
218                 buff.append(")");
219                 return buff.toString();
220         }
221
222         private void swap(int i, int j) {
223                 int t = p[i];
224                 p[i] = p[j];
225                 p[j] = t;
226         }
227 }
228
```

:::::::::::::::::::::::: **Point2D.java** ::::::::::::::::::::::::

```
1 import java.io.*;
2 import java.math.*;
3
4 public class Point2D implements Serializable{
5         public float x, y;
6
7         public Point2D() {
8                 x = 0;
9                 y = 0;
10              }
11
12         public Point2D(float a, float b) {
13                 x = a;
14                 y = b;
15         }
16
17         public void set(float a, float b) {
18                 x = a;
19                 y = b;
20         }
21
22         public Point2D(Point2D p) {
23                 x = p.x;
24                 y = p.y;
25         }
26
27         public void set(Point2D p) {
28                 x = p.x;
29                 y = p.y;
30         }
31
32         public double dist(Point2D p) {
33                 float dx = x - p.x;
34                 float dy = y - p.y;
35                 return Math.sqrt(dx*dx + dy*dy);
36         }
37 }
```

C.7 Mersenne Prime

The following example illustrates how mobile actors are used to search for mersenne primes in parallel. We limit our search to the range $2^{4000} - 1$ thru $2^{5000} - 1$; where $N = 1000$. The two known mersenne primes within this range are $2^{4253} - 1$ and $2^{4423} - 1$. The primality test is based on Lucas-Lehmer Test [Doc98]. All multiplication is done using a fast FFT algorithm. Given p machines, we decompose the problem such that each machine performs a test on a range of size N/p. Machines are organized using a master-slave design. Given p machines the master distributes the subcomputations to the $p-1$ slaves and waits for the result of the primality test from each slave. In addition, the master also performs a primality test. The decomposition process is done on Lines (38-45 *PrimeTest.java*). Remote actor creation is performed using the function *createRemote*(); actor placement is dynamic (Line 42 *PrimeTest.java*). After program decomposition, the current actor changes its state to itself in order to wait for the result (Line 47 *PrimeTest.java*). Upon receipt of a communication "result", the method *result* in the class "TSP" is executed. Results are collected as multiple inputs and use a strategy similar to the Parallel Sort/Sum algorithms (Section C.4 and C.5).

:::::::::::::::::::::::::: **PrimeTest.java** ::::::::::::::::::::::::::

```
1 import java.io.*;
2 import java.util.*;
3
4 import jmas.util.*;
5 import jmas.actor.*;
6
7 public class PrimeTest extends MobileActor
8 {
9         final int fromprime = 4000;
10         final int toprime = 5000;
11         final int dif = toprime - fromprime;
12
13         Long time;
14         int in;
15         Integer procs = null;
16
17         public PrimeTest() { }
18
19         public PrimeTest(Long t, Integer r, Integer c)
20         {
21              time = t;
22              in = r.intValue();
23              procs = c;
24         }
25
26         public void boot()
27         {
28              init(new Integer(2));
29         }
30
```

```
31          public void init(Integer p)
32          {
33               ActorAddress addr;
34               Object[] param;
35               int procs = p.intValue();
36               int numiter = dif/procs;
37
38               System.out.println("Mersenne Prime <" + procs + " procs>.....");
39               for(int i = 0,j = fromprime;i < procs;i++, j+= numiter)
40               {
41                    param = new Object[] {new Integer(j), new Integer((j+numiter)-1)};
42                    addr = createRemote("mersenne",getAvailableHost(), param);
43
44                    send(addr,"isPrime", getAddress());
45               }
46               time = new Long(System.currentTimeMillis());
47               become("PrimeTest", new Object[] {time, p, p});
48          }
49
50          public void result(String status)
51          {
52               if(status.equals("SINK"))
53                    --in;
54               else
55                    System.out.println(status);
56
57               if(in > 0)
58                  become("PrimeTest", new Object[] { time, new Integer(in),procs });
59               else
60               {
61                    long f = System.currentTimeMillis();
62                    System.out.println("Elapse Time: " +
                           (float)((f - time.longValue())/1000.0) + " secs\n\n");
63
64                    if(procs.intValue() != 15)
65                    {
66                      ActorAddress c = create("PrimeTest");
67                      if(procs.intValue() == 14)
68                        send(c,"init", new Integer(procs.intValue()+1));
69                      else
70                        send(c,"init", new Integer(procs.intValue()+2));
71                    }
72               }
73          }
74 }
```

Each subcomputation is an actor *mersenne* (*mersenne.java*). Upon receipt of a communication "isPrime", the actor performs a primality test on the given range of numbers. the total possible tours (Lines 575-614). No two actors search the same range of numbers. If a prime is found the result is returned to the master actor (Line 605 mersenne.java) and the search process continues. After performing the test a termination message "SINK" is sent to the master machine (Line 612 *mersenne.java*).

:::::::::::::::::::::::::: **mersenne.java** ::::::::::::::::::::::::::

```
1
2 /*  mersenne.java - Discrete Weighted Transform, irrational base method for
3               Lucas-Lehmer Mersenne test.
4
5     References:
6
7     Crandall R E and Fagin B 1994; "Discrete Weighted Transforms
8             and Large-Integer Arithmetic," Math. Comp. 62, 205, 305-324
9     Crandall R E 1995; "Topics in Advanced Scientific Computation,"
```

```
10              TELOS/Springer-Verlag
11  */
12 import java.io.*;
13 import java.lang.*;
14 import java.util.*;
15 import jmas.util.*;
16 import jmas.actor.*;
17
18 public class mersenne extends MobileActor
19 {
20  final static double  TWOPI  = (double)(2*3.14159265358979323846433);
21  final static double  SQRTHALF = (double)(0.70710678118654752400844362104);
22  final static double SQRT2 = (double)(1.41421356237309504880168724209);
23  final static int BITS = 16;
24
25  double[] cn, sn, two_to_phi, two_to_minusphi, scrambled;
26  double high,low,highinv,lowinv;
27  int b, c, start, stop;
28  int[]  permute;
29
30  public mersenne(Integer st, Integer sp)
31  {
32    start = st.intValue();
33    stop = sp.intValue();
34  }
35
36  public double rint(double x){      return((double)(long)(x+0.5));}
37
38  public void print(double[] x, int N)
39  {
40   int printed = 0;
41
42    while(N-- > 0) {
43     if ((x[N]==0) && (printed == 0)) continue;
44     System.out.print((long)(x[N]) + " ");
45     printed=1;
46     }
47    System.out.println("");
48  }
49
50  public void  init_scramble_real(int n)
51  {
52         int i,j,k,halfn = n>>1;
53         int tmp;
54
55         for(i=0; i<n; ++i) permute[i] = i;
56         for(i=0,j=0;i<n-1;i++) {
57                 if(i<j) {
58                   tmp = permute[i];
59                   permute[i] = permute[j];
60                   permute[j] = tmp;
61                 }
62                 k = halfn;
63                 while(k<=j) {
64                         j -= k;
65                         k>>=1;
66                 }
67                 j += k;
68         }
69  }
70
71  public void init_fft(int n)
72  {
73        int j;
74        double e = (double) (TWOPI/n);
75
76         cn = new double[n];
77         sn = new double[n];
78         for(j=0;j<n;j++) {
79                 cn[j] = Math.cos(e*j);
80                 sn[j] = Math.sin(e*j);
81         }
82
83         permute = new int[n];
84         scrambled = new double[n];
85         init_scramble_real(n);
```

```
86
87  }
88
89  public void fft_real_to_hermitian(double[] z, int n)
90  /* Output is {Re(z^[0]),...,Re(z^[n/2),Im(z^[n/2-1]),...,Im(z^[1]).
91     This is a decimation-in-time, split-radix algorithm.
92  */
93  {       int n4;
94          double[] x = new double[n+1];
95          double cc1, ss1, cc3, ss3;
96          int i1, i2, i3, i4, i5, i6, i7, i8,
97                      a, a3, dil;
98          double t1, t2, t3, t4, t5, t6;
99          double e;
100          int nn = n>>1, nminus = n-1, is, id;
101          int n2, n8, i, j;
102
103         System.arraycopy(z,0,x,1,n);
104         x[0] = 0.0;
105         is = 1;
106         id = 4;
107
108         do{
109            for(i2=is;i2<=n;i2+=id) {
110                 i1 = i2+1;
111                 e = x[i2];
112                 x[i2] = e + x[i1];
113                 x[i1] = e - x[i1];
114            }
115            is = (id<<1)-1;
116            id <<= 2;
117         } while(is<n);
118         n2 = 2;
119         nn >>= 1;
120         while(nn > 0) {
121                 n2 <<= 1;
122                 n4 = n2>>2;
123                 n8 = n2>>3;
124                 is = 0;
125                 id = n2<<1;
126                 do {
127                         for(i=is;i<n;i+=id) {
128                                 i1 = i+1;
129                                 i2 = i1 + n4;
130                                 i3 = i2 + n4;
131                                 i4 = i3 + n4;
132                                 t1 = x[i4]+x[i3];
133                                 x[i4] -= x[i3];
134                                 x[i3] = x[i1] - t1;
135                                 x[i1] += t1;
136                                 if(n4==1) continue;
137                                 i1 += n8;
138                                 i2 += n8;
139                                 i3 += n8;
140                                 i4 += n8;
141                                 t1 = (x[i3]+x[i4])*SQRTHALF;
142                                 t2 = (x[i3]-x[i4])*SQRTHALF;
143                                 x[i4] = x[i2] - t1;
144                                 x[i3] = -x[i2] - t1;
145                                 x[i2] = x[i1] - t2;
146                                 x[i1] += t2;
147                         }
148                         is = (id<<1) - n2;
149                         id <<= 2;
150                 } while(is<n);
151                 dil = n/n2;
152                 a = dil;
153                 for(j=2;j<=n8;j++) {
154                         a3 = (a+(a<<1))&(nminus);
155                         cc1 = cn[a];
156                         ss1 = sn[a];
157                         cc3 = cn[a3];
158                         ss3 = sn[a3];
159                         a = (a+dil)&(nminus);
160                         is = 0;
161                         id = n2<<1;
162                         do {
163                                 for(i=is;i<n;i+=id) {
164                                         i1 = i+j;
```

```
165                                         i2 = i1 + n4;
166                                         i3 = i2 + n4;
167                                         i4 = i3 + n4;
168                                         i5 = i + n4 - j + 2;
169                                         i6 = i5 + n4;
170                                         i7 = i6 + n4;
171                                         i8 = i7 + n4;
172                                         t1 = x[i3]*cc1 + x[i7]*ss1;
173                                         t2 = x[i7]*cc1 - x[i3]*ss1;
174                                         t3 = x[i4]*cc3 + x[i8]*ss3;
175                                         t4 = x[i8]*cc3 - x[i4]*ss3;
176                                         t5 = t1 + t3;
177                                         t6 = t2 + t4;
178                                         t3 = t1 - t3;
179                                         t4 = t2 - t4;
180                                         t2 = x[i6] + t6;
181                                         x[i3] = t6 - x[i6];
182                                         x[i8] = t2;
183                                         t2 = x[i2] - t3;
184                                         x[i7] = -x[i2] - t3;
185                                         x[i4] = t2;
186                                         t1 = x[i1] + t5;
187                                         x[i6] = x[i1] - t5;
188                                         x[i1] = t1;
189                                         t1 = x[i5] + t4;
190                                         x[i5] -= t4;
191                                         x[i2] = t1;
192                                 }
193                                 is = (id<<1) - n2;
194                                 id <<= 2;
195                         } while(is<n);
196                 }
197         nn >>= 1;
198         }
199         System.arraycopy(x,1,z,0,n);
200 }
201
202 public void fftinv_hermitian_to_real(double[] z, int n)
203 /* Input is {Re(z^[0]),...,Re(z^[n/2),Im(z^[n/2-1]),...,Im(z^[1]).
204    This is a decimation-in-frequency, split-radix algorithm.
205  */
206 {
207          int n4;
208          double cc1, ss1, cc3, ss3;
209          double t1, t2, t3, t4, t5;
210          double[] x = new double[n+1];
211          int n8, i1, i2, i3, i4, i5, i6, i7, i8, a, a3, dil;
212          double e;
213          int nn = n>>1, nminus = n-1, is, id;
214          int n2, i, j;
215
216         System.arraycopy(z,0,x,1,n);
217         x[0] = 0.0;
218
219         n2 = n<<1;
220         nn >>= 1;
221         while(nn > 0) {
222                 is = 0;
223                 id = n2;
224                 n2 >>= 1;
225                 n4 = n2>>2;
226                 n8 = n4>>1;
227                 do {
228                         for(i=is;i<n;i+=id) {
229                                 i1 = i+1;
230                                 i2 = i1 + n4;
231                                 i3 = i2 + n4;
232                                 i4 = i3 + n4;
233                                 t1 = x[i1] - x[i3];
234                                 x[i1] += x[i3];
235                                 x[i2] += x[i2];
236                                 x[i3] = t1 - x[i4] - x[i4];
237                                 x[i4] = t1 + x[i4] + x[i4];
238                                 if(n4==1) continue;
239                                 i1 += n8;
240                                 i2 += n8;
241                                 i3 += n8;
242                                 i4 += n8;
243                                 t1 = x[i2]-x[i1];
```

```
244                                 t2 = x[i4]+x[i3];
245                                 x[i1] += x[i2];
246                                 x[i2] = x[i4]-x[i3];
247                                 x[i3] = -SQRT2*(t2+t1);
248                                 x[i4] = SQRT2*(t1-t2);
249                         }
250                         is = (id<<1) - n2;
251                         id <<= 2;
252                 } while(is<nminus);
253                 dil = n/n2;
254                 a = dil;
255                 for(j=2;j<=n8;j++) {
256                         a3 = (a+(a<<1))&(nminus);
257                         cc1 = cn[a];
258                         ss1 = sn[a];
259                         cc3 = cn[a3];
260                         ss3 = sn[a3];
261                         a = (a+dil)&(nminus);
262                         is = 0;
263                         id = n2<<1;
264                         do {
265                            for(i=is;i<n;i+=id) {
266                                 i1 = i+j;
267                                 i2 = i1+n4;
268                                 i3 = i2+n4;
269                                 i4 = i3+n4;
270                                 i5 = i+n4-j+2;
271                                 i6 = i5+n4;
272                                 i7 = i6+n4;
273                                 i8 = i7+n4;
274                                 t1 = x[i1] - x[i6];
275                                 x[i1] += x[i6];
276                                 t2 = x[i5] - x[i2];
277                                 x[i5] += x[i2];
278                                 t3 = x[i8] + x[i3];
279                                 x[i6] = x[i8] - x[i3];
280                                 t4 = x[i4] + x[i7];
281                                 x[i2] = x[i4] - x[i7];
282                                 t5 = t1 - t4;
283                                 t1 += t4;
284                                 t4 = t2 - t3;
285                                 t2 += t3;
286                                 x[i3] = t5*cc1 + t4*ss1;
287                                 x[i7] = -t4*cc1 + t5*ss1;
288                                 x[i4] = t1*cc3 - t2*ss3;
289                                 x[i8] = t2*cc3 + t1*ss3;
290                            }
291                            is = (id<<1) - n2;
292                            id <<= 2;
293                         } while(is<nminus);
294                 }
295         nn >>= 1;
296         }
297         is = 1;
298         id = 4;
299         do {
300           for(i2=is;i2<=n;i2+=id){
301                 i1 = i2+1;
302                 e = x[i2];
303                 x[i2] = e + x[i1];
304                 x[i1] = e - x[i1];
305           }
306           is = (id<<1) - 1;
307           id <<= 2;
308         } while(is<n);
309         e = 1/(double)n;
310         System.arraycopy(x,1,z,0,n);
311         for(i=0;i<n;i++) z[i] *= e;
312 }
313
314 public void square_hermitian(double[] b, int n)
315 {
316          int k, half = n>>1;
317          double c, d;
318
319         b[0] *= b[0];
320         b[half] *= b[half];
321         for(k=1;k<half;k++) {
322                 c = b[k]; d = b[n-k];
```

```
323                 b[n-k] = 2.0*c*d;
324                 b[k] = (c+d)*(c-d);
325         }
326 }
327
328
329 public void squareg(double[] x,int size)
330 {
331    fft_real_to_hermitian(x, size);
332    square_hermitian(x, size);
333    fftinv_hermitian_to_real(x, size);
334 }
335
336 public void init_lucas(int q,int N)
337 {
338  int j,qn,a;
339  double log2 = Math.log(2.0);
340
341  two_to_phi = new double[N];
342  two_to_minusphi = new double[N];
343
344  low = rint(Math.exp(Math.floor((double)q/N)*log2));
345  high = low*low;
346  lowinv = 1.0/low;
347  highinv = 1.0/high;
348  b = q & (N-1);
349  c = N-b;
350
351  two_to_phi[0] = 1.0;
352  two_to_minusphi[0] = 1.0;
353  qn = q&(N-1);
354
355   for(j=1; j<N; ++j) {
356    a = N - (( j*qn)&(N-1));
357    two_to_phi[j] = Math.exp(a*log2/N);
358    two_to_minusphi[j] = 1.0/two_to_phi[j];
359    }
360 }
361
362 public double addsignal(double[] x, int N, int error_log)
363 {
364    int k,j,bj,bk,sign_flip,NminusOne = N-1;
365    double zz,w;
366    int  xptr = 0, xxptr = 0;
367    double hi = high, lo = low,
368        hiinv = highinv, loinv = lowinv;
369    double err, maxerr = 0.0;
370
371   bk  =  0;
372   for(k=0; k<N; ++k) {
373     if ((zz=x[xptr])<0) {
374       zz = Math.floor(0.5 - zz);
375       sign_flip = 1;
376       }
377     else {
378       zz = Math.floor(zz+0.5);
379       sign_flip = 0;
380       }
381     if(error_log != 0 ) {
382         if(sign_flip == 1) err = Math.abs(zz + x[xptr] );
                else err = Math.abs(zz - x[xptr]);
383         if(err > maxerr) maxerr = err;
384     }
385     x[xptr] = 0;
386     j = k;
387     bj = bk;
388     xxptr = xptr++;
389     do {
390          if(j==N) j=0;
391          if(j==0){xxptr = 0; bj = 0;  w = Math.floor(zz*hiinv);
392                   if(sign_flip == 1) x[xxptr] -= (zz-w*hi);
                          else x[xxptr] += (zz-w*hi); }
393          else if(j==NminusOne) { w = Math.floor(zz*loinv);
394                   if(sign_flip == 1) x[xxptr] -= (zz-w*lo);
                          else x[xxptr] += (zz-w*lo); }
```

```
395           else if (bj >= c) { w = Math.floor(zz*hiinv);
396                     if(sign_flip == 1) x[xxptr] -= (zz-w*hi);
                          else x[xxptr] += (zz-w*hi); }
397           else { w = Math.floor(zz*loinv);
398                     if(sign_flip == 1) x[xxptr] -= (zz-w*lo);
                          else x[xxptr] += (zz-w*lo); }
399           zz = w;
400           ++j;
401           ++xxptr;
402           bj += b; if(bj>=N) bj -= N;
403           }  while(zz!=0.0);
404      bk += b; if(bk>=N) bk -= N;
405      }
406      return(maxerr);
407  }
408
409  public void patch(double[] x,int N)
410  {
411    int j,bj,NminusOne = N-1;
412    long carry;
413    double hi = high, lo = low, highliminv, lowliminv, xx;
414    int px = 0;
415    double  highlim,lowlim, lim, inv, base;
416
417     carry = 0;
418     highlim = hi*0.5;
419     lowlim = lo*0.5;
420     highliminv =1.0/highlim;
421     lowliminv = 1.0/lowlim;
422
423     xx = x[px] + carry;
424     if (xx >= highlim) carry =((long)(xx*highliminv+1))>>1;
425       else if (xx<-highlim) carry = -(((long)(1-xx*highliminv))>>1);
426         else carry = 0;
427     x[px++] = (double)(xx - carry*hi);
428
429   bj = b;
430   for(j=1; j<NminusOne; ++j) {
431     xx = x[px] + carry;
432     if ((bj & NminusOne) >= c) {
433       if (xx >= highlim) carry =((long)(xx*highliminv+1))>>1;
434         else if (xx<-highlim) carry = -(((long)(1-xx*highliminv))>>1);
435           else carry = 0;
436        x[px] = (double) (xx - carry*hi);
437      } else {
438       if (xx >= lowlim) carry =((long)(xx*lowliminv+1))>>1;
439       else if (xx<-lowlim) carry = -(((long)(1-xx*lowliminv))>>1);
440         else carry = 0;
441       x[px] = (double) (xx - carry*lo);
442      }
443     ++px;
444     bj += b;
445   }
446
447     xx = x[px] + carry;
448     if (xx >= lowlim) carry = ((long)(xx*lowliminv+1))>>1;
449       else if (xx<-lowlim) carry = -(((long)(1-xx*lowliminv))>>1);
450         else carry = 0;
451     x[px] = (double) (xx - carry*lo);
452
453     if (carry != 0) {
454       j = 0;
455       bj = 0;
456       px = 0;
457       while(carry != 0) {
458         xx = x[px] + carry;
459         if (j==0) { lim = highlim; inv = highliminv; base = hi;}
460           else if (j==NminusOne) {lim = lowlim; inv = lowliminv; base = lo;}
461             else if ((bj & NminusOne) >= c) {lim = highlim; inv = highliminv;
462                       base = hi; }
463               else {lim = lowlim; inv = lowliminv; base = lo;}
464          if (xx>=lim) carry = ((long)(xx*inv+1))>>1;
465            else if (xx<-lim) carry = -(((long)(1-xx*inv))>>1);
466              else carry = 0;
```

```
467           x[px++] = (double) (xx - carry*base);
468           bj += b;
469            if (++j == N) {
470              j = 0;
471              bj = 0;
472              px = 0;
473              }
474        }
475      }
476  }
477
478  public  void check_balanced(double[] x,int N)
479  {
480   int j,bj = 0,NminusOne = N-1;
481   double limit, hilim,lolim;
482   int ptrx = 0;
483
484   hilim = high*0.5;
485   lolim = low*0.5;
486   for(j=0; j<N; ++j) {
487         if (j==0)  limit = hilim;
488           else if (j==NminusOne) limit = lolim;
489             else if ((bj & NminusOne) >= c) limit = hilim;
490               else limit = lolim;
491           if (!((x[ptrx]<=limit) && (x[ptrx]>=-limit)))
492              System.exit(1);
493           ++ptrx;
494           bj+=b;
495           }
496  }
497
498  public double lucas_square(double[] x,int N, int error_log)
499  {
500    int j, perm = 0, ptrx = 0, ptry = 0, ptrmphi = 0;
501    double err;
502
503   for(j=0; j<N; ++j, perm++)
504        scrambled[ptry++] = x[permute[perm]] * two_to_phi[permute[perm]];
505
506   squareg(scrambled,N);
507
508   perm = 0;
509   ptrx = 0;
510   ptrmphi = 0;
511
512   for(j=0; j<N; ++j)
513        x[ptrx++] = scrambled[permute[perm++]] *  two_to_minusphi[ptrmphi++];
514
515   err = addsignal(x,N, error_log);
516   patch(x,N);
517   if (error_log != 0) check_balanced(x,N);
518
519   return(err);
520  }
521
522  public int iszero(double[] x,int N)
523  {
524    int j;
525
526   for(j=0; j<N; ++j) if (rint(x[j]) != 0) return 0;
527   return 1;
528  }
529
530  public void balancedtostdrep(double[] x,int N)
531  {
532   int sudden_death = 0, j = 0, NminusOne = N-1, bj = 0;
533
534   while(true) {
535     if (x[j] < 0) {
536       --x[(j+1) & NminusOne];
537       if (j==0) x[j]+=high;
538         else if (j==NminusOne) x[j]+=low;
539           else if ((bj & NminusOne) >=c) x[j]+=high;
540             else x[j]+=low;
541       }
542     else if (sudden_death == 1) break;
543
```

```
544     bj+=b;
545     if (++j==N) {
546       sudden_death = 1;
547       j = 0;
548       bj = 0;
549       }
550     }
551   }
553
554   public void printbits(double[] x, int q, int N, int totalbits)
555   {
556    char[] bits = new char[totalbits];
557    int j, k, i, word;
558
559    j = 0;
560    i = 0;
561    do {
562       k = (int)( Math.ceil((double)q*(j+1)/N) - Math.ceil((double)q*j/N));
563       if (k>totalbits) k = totalbits;
564       totalbits -= k;
565       word = (int)x[j++];
566       while(k-- > 0) {
567         bits[i++] = (char) ('0' + (word & 1));
568         word>>=1;
569         }
570       } while(totalbits > 0);
571     while(i-- > 0) System.out.print(bits[i]);
572     System.out.println("");
573   }
574
575   public void isPrime(ActorAddress cust)
576   {
577    int q, n, j,i,k;
578    double[] x;
579    double w, err;
580    int last,errflag=0;
581
582      for(q = start; q < stop;q++)
583      {
584          last = q-1;
585          n = 256;
586
587          x = new double[n];
588          init_fft(n);
589           init_lucas(q,n);
590
591          for(j=0;j<n;j++) x[j]=0;
592          x[0] = 4.0;
593
594          for(j=1;j<last;j++) {
595            err = lucas_square(x,n,errflag);
596          if (errflag > 0)
597             System.out.println(j + " maxerr: " + err);
598            x[0]-=2.0;
599          System.out.println("iter: " + j);
600          }
602
603          System.out.println(q + " ");
604          if (iszero(x,n) == 1)
605              send(cust,"result",new String(q + "IS PRIME"));
606            else {
607               balancedtostdrep(x,n);
608              printbits(x,q,n,64);
609              System.out.println("");
610               }
611      }
612      send(cust,"result",new String("SINK"));
613      sink();
614    }
615  }
```

VITA

Legand L. Burge III

Candidate for the Degree of

Doctor of Philosophy

Thesis: JMAS: A JAVA-BASED MOBILE ACTOR SYSTEM FOR HETEROGENEOUS DISTRIBUTED PARALLEL COMPUTING

Major Field: Computer Science

Biographical Data:

Personal Data: Born in Stillwater, Oklahoma on February 5, 1972, the son of Dr. Legand L. Burge Jr. and Gwenetta V. Burge

Education: Graduated from John Marshall High School, Oklahoma City, Oklahoma, 1989; received Bachelor of Science in Computer Science/Mathematics from Langston University, Langston, Oklahoma in 1992. Receive the Master of Science in Computer Science from Oklahoma State University in July 1995. Completed the requirements for the Doctor of Philosophy in Computer Science at Oklahoma State University in December 1998.

Experience: *Research Assistant*, Oklahoma State University, Department Oklahoma State University 1992 to 1998. *Adjunct Instructor*, Langston University, Department of Computer and Information Science 1993 to 1998. *Software Engineer*, Teubner and Associates, Stillwater, Oklahoma 1995. *Computer Analyst*, National Security Agency, Ft. George G. Meade, Maryland, 1991 to 1995.