5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)

San Diego, California, USA, May 3–7, 1999

JMAS: A Java-Based Mobile Actor System for Distributed Parallel Computation

Legand L. Burge III
Howard University

K. M. George
Oklahoma State University

JMAS: A Java-Based Mobile Actor System for Distributed Parallel Computation

Legand L. Burge III [*]
Systems and Computer Science
Howard University
Washington, DC 20059
blegand@scs.howard.edu

K. M. George
Computer Science Department
Oklahoma State University
Stillwater, Oklahoma 74078
kmg@a.cs.okstate.edu

Abstract

JMAS is a prototype network computing infrastructure based on mobile actors [10] using Java technology. JMAS 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. Applications are decomposed by the programmer into small, self-contained sub-computations and distributed among a virtual network of Distributed Run-Time Managers (*D-RTM*); which execute and manage all mobile computations. This system is well suited for course grain computations for network computing clusters. Performance evaluation is done using two benchmarks: a Mersenne Prime Application, and the Traveling Salesman Problem.

Keywords: Distributed systems, parallel computing, actor model, mobile agents, actors, network computing

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. 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 [7, 30]. The combined resources of millions of computers on the Internet can be harnessed to form a powerful global computing infrastructure consisting of workstations, PCs, and supercomputers (Figure 1).

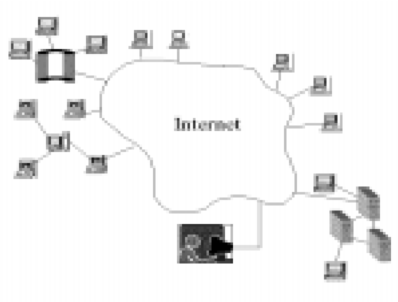

Figure 1 Global Computing Infrastructure.

The vision of integrating network computers into a global computing resource is as old as the Internet[7][23]. 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 [29], MPI [24], PVM [31], Piranha [22], NEXUS [28], Network of Workstations (*NOW*) [3], Legion [23], and GLOBUS [20]. These network computing frameworks use low-level communication systems, or 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, 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. Recently, researchers have proposed several

[*]This work was supported in part by the USENIX Student Research Grant

approaches to provide a platform independent Java-based high-performance network computing infrastructure. These include Javalin [16], WebFlow [4], IceT [14], JavaDC [15], Parallel Java [26], Parallel Java Agents [27], ATLAS [5], Charlotte [6], ParaWeb [9], Popcorn [12], and Ninflet [32]. 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. [13], Fox et al. [21] and implemented in [33]. 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.

Mobile agents are a convenient paradigm for distributed computing [8, 18]. 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. Mobile agents allow a distributed application to be written as a single program.

In this paper we discuss the design and implementation of a prototype network computing system (*JMAS*) based on the mobile actor model [10] using Java technology [25]. The mobile actor model is a parallel programming paradigm for distributed parallel computing based on mobile agents and the *actor* message passing model [1]. 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 network computing environment. Our experiments were ran using two benchmarks: a Mersenne Prime Application, and the Traveling Salesman Problem.

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. 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. 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;

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.

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. 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. We present a communication paradigm among mobile agents that incorporates actor-based message passing to support dynamic architecture topologies for distributed parallel computing.

3 The Mobile Actor Paradigm

A *mobile actor* is an 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. 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. The mobile actor model is 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. The execution of computations may migrate across file systems consisting of networks of computers and/or computing clusters.

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 state change.
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.

4 JMAS: A Java-Based Mobile Actor System

Exploiting the resources of several interconnected computers to form a powerful network 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 network computing infrastructure which executes mobile actor computations. A mobile actor system is a multi-user, heterogeneous, network 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,
- be fault tolerant, and
- be secure.

4.1 JMAS Infrastructure

JMAS is a network computing environment for executing mobile actor computations. JMAS is designed using Java technology [25], 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 Language Support in JMAS

JMAS is based on the Java Programming Language and Virtual Machine of JDK1.1 [25]. JDK1.1 contains mechanisms that allow objects to be read/written to streams (object serialization), 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 lightweight 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 send communications to an actor.

4.3 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 [11], that forwarding messages in a distributed system consisting of N machines requires in the worst case $N-1$ message rounds.

4.4 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 local 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 [12]. 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*. Computations are distributed to seller using a round-robin schedule. This strategy is intended for coarse-grain applications.

4.5 Security in JMAS

Security issues are not addressed in this version of the 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.

4.6 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. Further, if a host should fail, then JMAS will remove that host from the current CPU Market configuration.

5 JMAS Architecture

The architecture of JMAS is organized as a series of layers or levels, each one built upon its predecessor (Figure 2). 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).

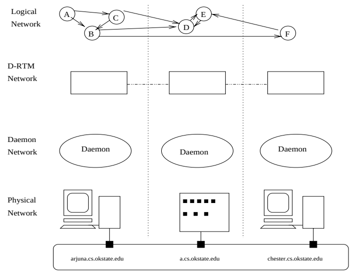

Figure 2. Four Layer Mobile Actor Architecture.

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) [3]. 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 interconnected networks of computer clusters.

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 3).

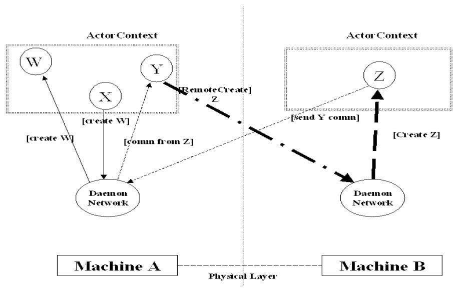

Figure 3. Message-driven model of execution.

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.

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 network computing system. The D-RTM was designed using a layered virtual machine design built on top of the Java Virtual Machine (JVM) using JDK1.1 [25] (Figure 4). 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 preemptive, priority schedule (i.e. **Scheduler**),

- To manage the CPU load on the local machine (i.e. **Load Balancer**).

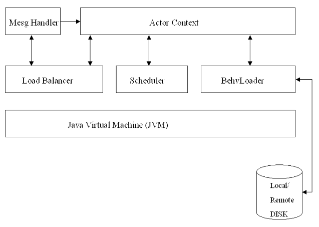

Figure 4. Distributed Run-Time Manager (D-RTM).

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, 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.

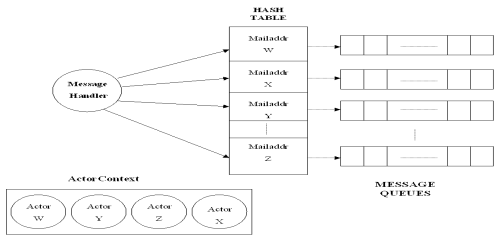

Figure 5. Message Handler.

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 [25]. The classes needed to perform these operations are obtained from the *java.lang.reflect* package of the JDK1.1.

5.3.3 Scheduler

JMAS implements a pre-emptive, priority-based scheduler among local 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 6(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 6(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. Computations are migrated to remote locations using a round-robin schedule.

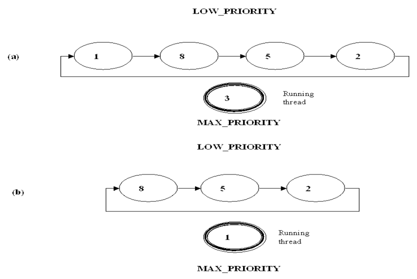

Figure 6. Thread Scheduler.

5.3.4 ClassLoader

In order 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 7).

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.

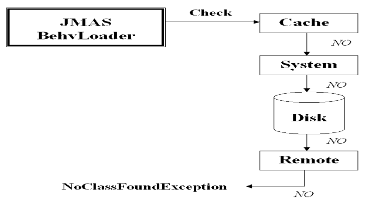

Figure 7. Operation of JMAS ClassLoader.

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 [12]. 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.

5.3.5.1 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 8, 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 8). 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.

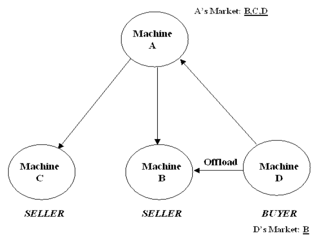

Figure 8. CPU Market Hierarchy.

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 9, 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 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 [11].

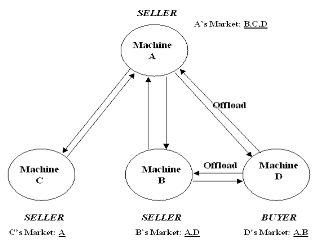

Figure 9. Host A Notifies Markets of B,C, and D.

5.3.5.2 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 10, the Load Balancer maintains a load below 75% of the threshold, and 25% of the threshold above the minimum load (i.e. zero).

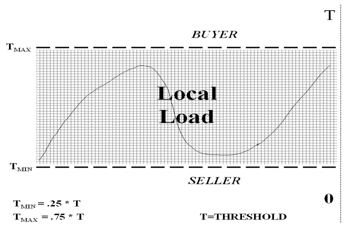

Figure 10. Load Balancing Policy.

Before starting a thread on the local machine, the load balancer checks the current load to insure that it 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 11). 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. By default the status of a machine is *seller.* Therefore this field is changed to status *buyer.*

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 11. 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. 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. 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.

6 Performance Evaluation

JMAS offers the basic infrastructure needed to integrate computers connected by a network 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 [17],[19]. We conducted our experiments in an 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.
- 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.

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 network computing 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 network computing environment were communication cost is high, the number and packet size of inter-process communications must be limited. Table 1 shows the performance of two microbenchmarks 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 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 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.

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

Using the equation above, we can estimate the performance of a given application. 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 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 network computing 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 absolute minimum execution time for the continuous function $T_{Cost}(p)$ on a closed bounded interval $[1, p]$; where $p = t_{seq}/(t_{load} + k * t_{send})$. Giving,

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

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

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

Therefore, we can estimate the maximum performance in 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,

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

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 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.

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 time.

6.4 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 $Perm()$ 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 12). Processors are arranged in a master-slave design.

```
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 lexicograph-
    ical 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
    Perm(i) : Generates the ith tour in lexicographical order

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 Perm(i) distance ≤ Perm(mintour) distance
             set mintour to i
  4. send mintour to cust

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 ad-
         dress 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 Perm(resultTour) distance ≤ Perm(mintour) dis-
         tance
             set mintour to resultTour
```

Figure 13. TSP Algorithm.

6.4.1 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 13, 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 14 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 a better performance. 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 2, the average CPU utilization for the best possible number of machines p is 50%. This is because, 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 15). The estimates are also reflected in Figure 13. These results show that our framework is well suited for course grain applications. The TSP application also scales well to large computation sizes (Figure 16).

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 2. Estimating the Performance of TSP.

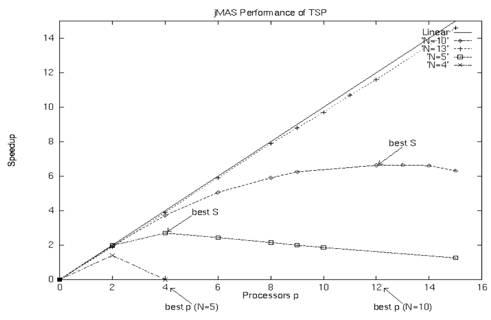

Figure 13. Speedup of TSP.

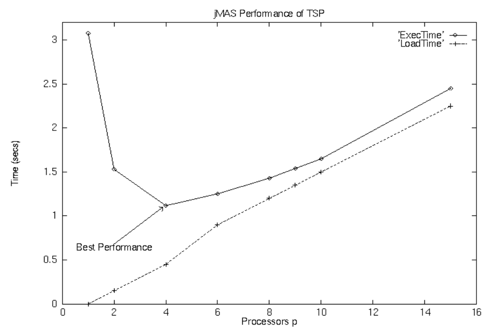

Figure 14. Execution Time vs Load Time.

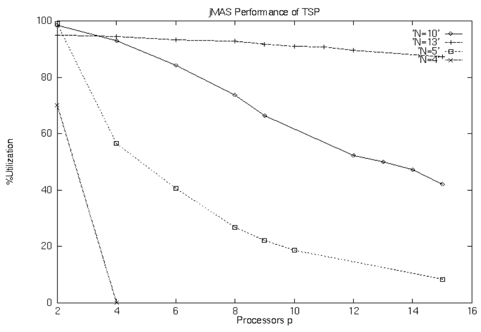

Figure 15. CPU Utilization of TSP.

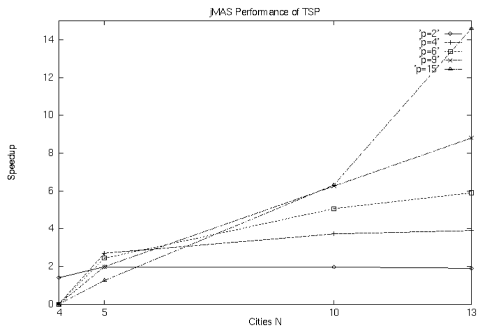

Figure 16. Scalability of TSP.

6.5 Mersenne Prime Application

For our second application, we implemented a parallel primality test which is used to search for Mersenne prime numbers [19]. 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 [19]. 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.

6.5.1 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 [19].

We develop a mobile actor program to test for Mersenne primality, given a range of prime numbers (Figure 18). 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.

```
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 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

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
```

Figure 17. Mersenne Prime Algorithm.

6.5.2 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 18, 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.

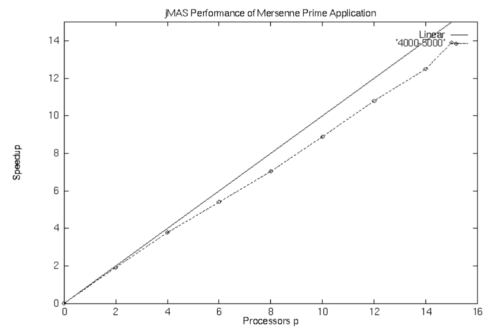

Figure 18. Speedup of Mersenne Prime.

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 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 19). 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 3. Estimating the Performance of the Mersenne Prime Test.

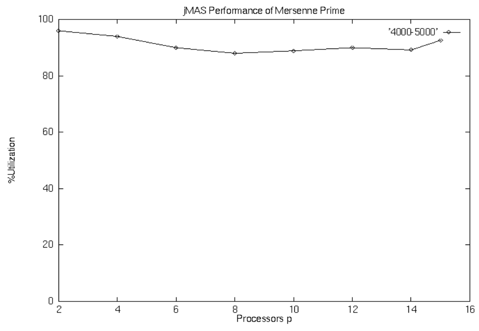

Figure 19. CPU Utilization of Mersenne Prime.

7 Conclusion

In this paper we discuss the design and implementation of a prototype network computing system (*JMAS*) based on the mobile actor model [10] using Java technology [25]. JMAS 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

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

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:

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

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

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

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.

7.1 Future Work

Issues such as fault tolerance and security need to be addressed and implemented within the JMAS framework. Also, experiments concerning the scalability of the JMAS framework to support internet (global) computing will be conducted in future work. Support for high-level communication abstractions will be addressed within the JMAS Mobile Actor API. Examples are barrier actors, mutex actors, call/return communication, and actorSpaces [2].

References

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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

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

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

[26] 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.

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

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

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

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

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

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

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