University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Journal Articles | Computer Science and Engineering, Department of

11-1-2003

Middleware Infrastructure for Parallel and Distributed Programming Models in Heterogeneous Systems

Jameela Al-Jaroodi
University of Nebraska - Lincoln, jaljaroo@cse.unl.edu

Nader Mohamed
University of Nebraska - Lincoln, nmohamed@cse.unl.edu

Hong Jiang
University of Nebraska - Lincoln, jiang@cse.unl.edu

David Swanson
University of Nebraska - Lincoln, dswanson@cse.unl.edu

Follow this and additional works at: http://digitalcommons.unl.edu/csearticles

Part of the Computer Sciences Commons

Al-Jaroodi, Jameela; Mohamed, Nader; Jiang, Hong; and Swanson, David, "Middleware Infrastructure for Parallel and Distributed Programming Models in Heterogeneous Systems" (2003). *CSE Journal Articles.* Paper 60.
http://digitalcommons.unl.edu/csearticles/60

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Journal Articles by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

Digital Object Identifier: 10.1109/TPDS.2006.18

Middleware Infrastructure for Parallel and Distributed Programming Models in Heterogeneous Systems

Jameela Al-Jaroodi, *Student Member*, *IEEE*, Nader Mohamed, *Student Member*, *IEEE*, Hong Jiang, *Member*, *IEEE*, and David Swanson

Abstract—In this paper, we introduce a middleware infrastructure that provides software services for developing and deploying high-performance parallel programming models and distributed applications on clusters and networked heterogeneous systems. This middleware infrastructure utilizes distributed agents residing on the participating machines and communicating with one another to perform the required functions. An intensive study of the parallel programming models in Java has helped identify the common requirements for a runtime support environment, which we used to define the middleware functionality. A Java-based prototype, based on this architecture, has been developed along with a Java Object-Passing Interface (JOPI) class library. Since this system is written completely in Java, it is portable and allows executing programs in parallel across multiple heterogeneous platforms. With the middleware infrastructure, users need not deal with the mechanisms of deploying and loading user classes on the heterogeneous system. Moreover, details of scheduling, controlling, monitoring, and executing user jobs are hidden, while the management of system resources is made transparent to the user. Such uniform services are essential for facilitating the development and deployment of scalable high-performance Java applications on clusters and heterogeneous systems. An initial deployment of a parallel Java programming model over a heterogeneous, distributed system shows good performance results. In addition, a framework for the agents' startup mechanism and organization is introduced to provide scalable deployment and communication among the agents.

Index Terms—Distributed systems middleware, parallel programming models, parallel and distributed Java, cluster, heterogeneous systems, distributed agents.

1 Introduction

A middleware infrastructure, using distributed software agents, is developed to provide services for high-performance programming environments and applications on clusters and networked heterogeneous systems. The agents enhance expandability, allowing the number of machines involved to grow easily by providing services that include job distribution, monitoring, and controlling for the system. This provides flexibility and ease of managing the different resources available. In addition, the distributed agents provide the required runtime support for the parallel and distributed applications developed at the application level.

A new choice of architecture for high-performance computing has emerged to be the cluster of PCs/workstations [9] and networked heterogeneous systems. Such systems provide processing power comparable to special purpose high end multiprocessor systems for a fraction of the cost. However, these systems require system and software services (middleware) that can support the distributed architectures and provide transparent and efficient utilization of the multiple machines. Moreover, as Java becomes more popular and available on many platforms, there is an increasing interest in using Java for high-performance computing and distributed applications on clusters due to its unique advantages in portability, security mechanisms, network programming capabilities, etc. This interest in Java has generated intensive research in Java for parallel, distributed, and multithreaded programming environments and tools on distributed systems.

The main objective of the paper is to identify the common requirements for the parallel and distributed programming models and to propose and design a middleware infrastructure to satisfy these requirements. In addition, the paper discusses a framework for the distributed agents' organization, configuration, and communication mechanisms to provide efficient, flexible, and scalable system support. Requirements such as remote loading and execution, resource management and scheduling, naming, security, group management and communications, and synchronization mechanisms were identified. Furthermore, the middleware infrastructure is designed to satisfy these requirements in a multilayered modular manner, which separates the programming model's specific functionalities from the general runtime support required by any parallel or distributed programming model. The layered approach also allows for easy modifications and updates of the different functions and services at the different layers and provides flexible component-based plug-ins. Therefore, the individual details of the middleware infrastructure components such as the scheduler,

• *The authors are with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, 115 Ferguson Hall, Lincoln, NE 68588-0115.*
E-mail: {jaljaroo, nmohamed, jiang, dswanson}@cse.unl.edu.

Manuscript received 8 Dec. 2002; accepted 31 Mar. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 118748.

1045-9219/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

resource manager, etc., can be separately considered as plug-in components. Moreover, the pure Java infrastructure based on distributed memory model provides portability, security, and the ability to utilize heterogeneous systems.

In the rest of this paper, Section 2 reviews related work and concepts. Then, we discuss parallel Java programming models and identify common infrastructure service requirements on clusters and heterogeneous systems in Section 3. In Section 4, we describe the architecture and features of the infrastructure and introduce the agent start-up organization and communication mechanisms in Section 5. Section 6 presents an example, the Java Object-Passing Interface (JOPI), that utilizes the middleware, along with the experimental evaluation of the performance. Finally, Section 7 concludes the paper with remarks about the main features and advantages of the middleware infrastructure and the current and future work.

2 Concepts and Related Work

Java's popularity among developers is increasing steadily and many research groups are exploring the possibilities of using Java for high-performance parallel computing on multiprocessor systems. Since Java is machine independent, the same Java programs can run on any platform with a Java virtual machine (JVM), without recompilation for each platform. In addition, Java is constantly being improved and optimized for performance. Very recently, research groups have worked on providing parallel Java using different approaches and programming models. This section introduces related concepts and lists some related projects.

2.1 Concepts

Java, in its current state, provides features and classes that facilitate distributed application development. Some of the features are object serialization [10], remote method invocation (RMI) [27], class loaders, network programming and Sockets, and the reflection API [10]. However, the development process of parallel applications in Java is complex and time consuming. Using the currently available methods, a daring programmer may be able to write a parallel application in Java, but the complexity of the task deters almost all from tackling this intricate task. On the other hand, the message-passing interface (MPI) [29] has provided languages such as C and FORTRAN with slightly simpler APIs to write parallel programs. Other MPI-based APIs such as OOMPI [28], Charm++ [22], and ABC++ [5], provide object-oriented message-passing interfaces.

2.2 Related Work

Many projects investigating parallel Java capabilities are in the research phase. Extensive literature study led us to classify them into the following four different categories based on how they provide parallelism, their compatibility with the JVM, and user involvement.

1. Java dialects and preprocessors. Projects such as Titanium [30] and HPJava [12] provide Java dialects for parallel programming with their own compilers, while others such as JAVAR [6] and JAVAB [7] provide parallelizing precompilers.
2. Alternatives to JVM. These projects provide parallel Java capabilities by altering the JVM or building a new one. Examples include JPVM [16] and cJVM [4].
3. Parallelizing multithreaded applications. Mechanisms that enable multithreaded applications to transparently utilize the underlying multiprocessor hardware are incorporated in projects of this category. This approach requires that the system distribute the threads among the distributed processors without user involvement. Representative projects include cJVM [4], JavaParty [26], and ProActive [11].
4. Pure Java implementations. Projects in this category provide parallelization facilities in pure Java implementations, which make the system portable and machine independent. Such systems require class libraries to provide the APIs needed to write parallel Java applications. ParaWeb [8], Ajents [20], Babylon [19], and JOPI [23] are some of the examples.

Our literature review [2] revealed that many research groups are working toward providing tools and programming models for parallel Java. Many of the projects provide message-passing interfaces based on MPI and MPI for Java (MPJ) draft specifications [13]. However, our approach to providing parallel-programming capabilities in Java has several differences from the projects studied. One significant difference is the separation of the enabling mechanisms, i.e., the middleware infrastructure, from the parallel programming models resulting in many advantages:

1. The infrastructure supports different programming models such as message-passing, object-passing, and distributed shared object.
2. The infrastructure provides efficient common services needed by any programming model such as scheduling, monitoring, load balancing, synchronization, and job control.
3. The programming models can be easily changed, upgraded, or completely reengineered without having to change the underlying support mechanisms.
4. The organization of the infrastructure and its close relationship with the models provides the flexibility to optimize and fine-tune its operations to achieve good performances.

3 Parallel Java on Heterogeneous Systems

Based on our observations from studying the different approaches for parallel programming in Java, we have identified some common requirements. In this section, we first discuss the different parallel Java programming models and study the requirements to implement and deploy these models, then identify the generic services and functions that the middleware should provide for developing and supporting the different programming models.

3.1 Java Parallel Programming Models

Providing parallel programming capabilities in Java can be achieved by following the known parallel programming models. These models are divided into four layers (categories) based on the level of user involvement in the

parallelization process and the achievable levels of efficiency. In addition, the implementation dependencies can be observed among these layers. They are described below in decreasing level of user involvement and system efficiency.

1. Message Passing: In this category, the system provides some form of information exchange mechanism among distributed processes. It provides, for example, functions to exchange messages among processes with point-to-point and group communication primitives, synchronization, and other operations. This programming model handles the remote process deployment and message exchange among the participating machines. The runtime support can be implemented as an independent middleware layer, thus providing a flexible and expandable solution. On the other hand, it can be implemented as an integral component of the model, thus making it more tightly coupled with the required functionality. However, the first approach is more flexible, expandable, and can be easily enhanced to support other models. The message-passing library and runtime support can be implemented in different ways such as pure Java implementations based on socket programming, native marshaling, and RMI [27], or by utilizing Java native interface (JNI), Java-to-C interface (JCI), parallel virtual machine (PVM), and other parallel infrastructures. A number of projects tried to comply with MPI [29] and MPJ [13], while others were based on a new set of APIs. Models in this category provide an efficient parallel programming environment because they directly utilize the basic communication mechanisms available; however, they are the least user friendly and require full user awareness of the parallelization process.
2. Distributed Shared Address Space: In the distributed shared address space or distributed shared object (DSO) model, the model presents an illusion to the user of a single address space where all or some data/objects are available to all participating processes. To provide this illusion, the programming model should be able to transparently handle all data/object communication, sharing, and synchronization issues, thus freeing the user from the concerns of operational details. One method of implementation is to utilize an available message-passing infrastructure. However, the programming model should handle the different issues of shared space such as information (data or objects) integrity and coherence, synchronization, and consistency. This category provides a more friendly development environment of parallel applications; however, the performance is penalized due to the overhead imposed by the sharing (coherence and consistency) and synchronization requirements.
3. Automatic Parallelization of Multithreaded Applications: This category aims to provide seamless utilization of a distributed environment to execute multithreaded applications on multiple machines. The main goal is to execute concurrent multithreaded applications in parallel without modifications. In this case, the implementation issues are similar to those in the distributed shared address space model in the sense that all data and objects used by more than one thread need to be sharable. As a result, the programming model requires data sharing or data exchange mechanisms to provide thread distribution and information sharing. To implement this model, a message-passing system or a DSM/DSO system can be used as the underlying support mechanisms. Such system is less efficient than a message passing due to the additional overhead of handling remote thread deployment, sharing, and synchronization.
4. Transparent (Automatic) Parallelization: Here, the goal is to execute sequential applications in parallel on multiple machines. Some systems provide transparent parallelization of Java programs written in standard Java by modifying the JVM, while others utilize preprocessors. The first approach introduces a new JVM that recognizes the existence of multiple machines and utilizes them to execute the Java bytecode in parallel. This JVM should handle the distribution of load and data/objects and efficiently utilize available resources. However, such JVM may be inefficient since many sequential applications are not easily parallelizable, especially if they were designed without parallelization in mind. Using preprocessors usually involves restructuring the existing Java code or bytecode to a parallel format. This process could be done either by parallelizing substructures such as loops or by introducing a different parallelization model such as message-passing or DSM in the code. In both cases, the main goals are to relieve the developer of the burden of explicitly parallelizing the application and to run current applications in parallel without (or with minor) modifications. Again, the programming model should be able to execute the automatically generated parallel programs. This model could be built from scratch or by utilizing any of the programming models described above. However, the efficiency achieved is not very high because applications vary and the systems cannot handle all of them at the same level of efficiency.

In addition to these four categories, a few research groups have also used combinations of these models or selected functionalities to provide different methods of parallelization. Although the message-passing model is the most difficult from a user's perspective, it is usually the most efficient because it is directly based on the system's basic communication mechanisms. However, the automatic parallelization is still the most attractive option for users since it does not involve any effort from them. Nevertheless, it is very difficult to achieve and the existing systems are not efficient. Detailed information about the classification, implementations, and comparison of parallel Java projects for heterogeneous systems can be found in [2].

3.2 Identifying Common Requirements

Standard Java technology such as JVM and JINI [15] provide a variety of features to develop and implement distributed Java applications. However, there are some key

features lacking when directly applied as an underlying infrastructure for constructing and executing parallel Java applications on heterogeneous systems. Some of these features are summarized as follows:

1. *Loading User Programs onto the Remote JVMs on the Participating Machines*: Java does not provide mechanisms to remotely load user classes on more than one JVM in parallel. A parallel application needs to be loaded onto JVMs of all nodes where it is scheduled. Thus, the parallel Java environment needs mechanisms to remotely load classes onto the selected nodes before starting the execution of the parallel application.
2. *Managing Resources and Scheduling User Jobs*: In order to efficiently run parallel applications, the system needs to schedule user programs based on the availability of the nodes and the available resources in each node. Thus, a mechanism is needed to monitor, manage, and maintain the resources of the entire cluster(s). Resources on a node may include the number of idle or sharable processors, memory space, current workload, the number of communication ports, and sharable data and objects.
3. *Security*: Some resources on the cluster nodes or distributed system may need to be protected from remote jobs being executed locally. For example, while a user is allowed to run a program on a remote node, he/she should not be allowed to access any files on the remote nodes or change their system properties without proper authorization. Although a basic security mechanism is available in Java to protect selected node resources, it would nevertheless be preferable if advanced security functions were available so that access control and security protocols can be easily defined and enforced through the middleware.
4. *Job and Thread Naming*: For an environment supporting multiple parallel jobs, a unique job ID needs to be assigned to each active job, which is needed to control a specific job, for example, to kill a job. In addition, each parallel job consists of multiple threads distributed among the nodes; therefore, a thread ID is needed for each thread. The thread ID is used to distinguish threads and control the flow of the parallel programs such as in message passing. For user threads to communicate, a mechanism to provide a mapping between logical thread IDs and actual network addresses such as IP address and network port is needed.
5. *User Commands*: Users need commands to submit, execute, and monitor their parallel programs and to control the environment from a single point on the cluster. Examples of these commands are to check available resources and currently running parallel jobs. These commands should provide the user with a single system image.
6. *Synchronization Mechanisms*: Any parallel application requires some form of synchronization and control to function correctly. Executing parallel applications on distributed environments makes these needs even more important. Basic mechanisms to ensure mutual exclusion, ordered execution, and barriers are necessary for many programming models. For example, a distributed shared object model will require synchronization to achieve consistency and coherence and an object-passing model requires explicit APIs for methods such as the barrier.
7. *Group Management and Communication*: A distributed parallel application requires collective communications at two different levels: at the job (or task) level to deploy, monitor, and control users jobs, and at the process level for interprocess communications such as broadcast and multicast. A programming model can benefit from the middleware for both levels, where efficient group communications methods can be utilized. Distributed applications may also require mechanisms to manage and control real-time dynamic agent and process membership in the system.

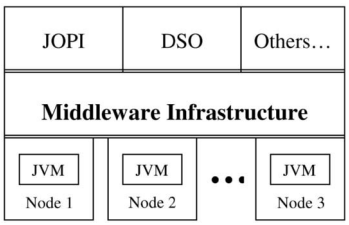

Fig. 1. The middleware infrastructure as runtime support for parallel and distributed programming models and applications on heterogeneous systems.

These common requirements can be implemented in different ways to provide the necessary tools and APIs for the programming model developer to build any of the aforementioned programming models. However, each model will also have its own set of functionalities that need to be implemented as part of the programming model itself. For example, in a distributed shared memory or object model, issues such as coherence and consistency must be handled within the programming model and independently from the middleware, while in a message passing model, it is left for the application developer to handle. In addition, some programming models can implement some functions already available in the middleware to achieve specific goals. For example, the communications functions in a message-passing model can be realized using the middleware functions or directly in the model to support specialized communications services that come with advanced cluster networks such as the Sockets-GM for Myrinet [25].

4 The Middleware Infrastructure

The middleware infrastructure is designed to satisfy the requirements discussed above. This system provides a pure Java infrastructure based on a distributed memory model, which makes it portable, secure, and capable of handling different programming models (see Fig. 1) such as JOPI [23] and the DSO model. The system has a number of components that collectively provide middleware services, including some of the requirements described above, for a high-performance Java environment on cluster and heterogeneous systems.

4.1 Agents

Software agent technology has been used in many systems to enhance the performance and quality of their services [18]. Our middleware infrastructure utilizes software agents

to provide flexible and expandable middleware services for high-performance Java environments. The main functions of the agents are to deploy, schedule, and support the execution of the parallel/distributed Java code, in addition to managing, controlling, monitoring, and scheduling the available resources on a single cluster or on a collection of heterogeneous systems. When a parallel Java application is submitted, an agent performs the following tasks:

1. Examine available resources and schedule the job for execution, while balancing the load.
2. Convert scheduled user classes into threads, then remotely upload and execute them directly from the main memories on the remote machines.
3. Monitor and control resources and provide monitoring and control functions to the user.

For high throughput, the agents are multithreaded, where each thread serves a client's request. Once user threads are deployed, they directly communicate with one another to perform parallel tasks, thus freeing the agents and reducing the overhead on the user programs. Agents' communication mechanisms are implemented using sockets and each agent consists of a number of components whose main functions are described below, although many of these functions can be independently enhanced to provide different levels of services:

1. The Request Manager handles user job requests such as deploying classes, starting/stopping a job, and checking agents/threads status. Requests come as request objects from client services or from other agents.
2. The Resource Manager provides methods to manage, schedule, and maintain the resources of the machine where the agent resides. It keeps records of executing threads, machine and communication resources' utilization, and performance information. In addition, it is responsible for reclaiming system resources after each job's completion or termination.
3. The Security Manager provides security measures for the system (see Section 4.3 for details).
4. The Class Loader remotely loads user classes in parallel onto the JVMs on the remote machines in preparation for execution.
5. The Scheduler selects the machines to execute a user job based on the requested number of processors. One mechanism to generate a schedule is to execute a test program to select the fastest responding machines. This method provides a simple but basic load balancing among the processors. However, since this is an independent component, the scheduler can be easily replaced by any suitable scheduler to satisfy different policies and performance requirements.

4.2 Client Services and Environment APIs

The client services and environment APIs provide commands for users to interact with the environment. Requests are accepted from the user and passed to the agent after encapsulation as an object with the necessary information. The following commands are available for the user through client services and for the other programming models and applications as APIs such as *pjava* to initiate a parallel job, *pingAgent* to list available agent(s) and their status, *listThreads* to list active threads, and *killJob* to terminate a job.

The client services class uses two types of classes for communication between clients and agents and among agents. The *agentClient* provides APIs to manage, control, and send requests for an agent and it is used for direct communication between the client and a given agent or among agents. In addition, the *agentGroup* provides APIs to manage, control, and send requests for a group of agents using the *agentClient* to individually communicate information to all agents in the group. For example, when a job is initiated, the request and schedule objects are passed to the *agentGroup*, which uses the *agentClient* to pass them to individual agents. Both *agentClient* and *agentGroup* are also used as API for developing distributed applications. When a programming model is developed using the runtime support environment, the interprocess communications are handled in different ways. Point-to-point communications, for example, can be implemented directly by the programming model. However, if the nodes/machines involved are not within a single cluster, the agents can assist the communications by providing routing mechanisms between the different nodes. In addition, group communications such as broadcast and multicast can be provided by the runtime environment rather than the programming model to achieve efficient distribution and response times.

4.3 Multiuser and Security Issues

The system allows multiple users to execute multiple jobs simultaneously. To properly manage these jobs, each job has multiple levels of identification, starting with a unique job ID assigned by the system. The user ID and the program name further distinguish different jobs. Within each job, thread IDs are used to identify the remote threads of the job. Executing user threads on remote machines exposes these machines to many "alien" threats, raising security and integrity concerns. Therefore, these machines must be protected to ensure safe execution. Java's default security manager provides some level of protection by checking operations against defined security policies before execution. However, the security manager in Java has some restrictions, thus many functions have been modified or rewritten for our system. More specifically, two modes of execution are used to provide a robust and secure environment:

1. **The Agent Mode** in which no restrictions are imposed. A thread running in this mode has full control of all the resources and operations in the system. All agents run in agent mode.
2. **The User Mode** in which restrictions are applied to limit the user access to the system resources. Some operations, such as deleting files, creating a subprocess, using system calls, modifying system properties, and writing files, are disabled in this mode.

With the security modes in place, the user processes have full access to resources on their local machines (where the user job was initiated), but limited and controlled access to all remote machines' resources (since they are running in user mode). To provide users with access to necessary resources for their application, the root (master) process executes on the user's local machine. However, the user has

the option to override this setting and allow the root process to execute on a remote machine; however, the application will have limited access to the system's resources. Nevertheless, this policy can be adapted to provide different levels of access control on the available machines. For example, a user on a cluster is given full access to all cluster nodes, but limited access to external systems. Another example is deploying an authentication/authorization policy for different access modes on the participating machines.

5 Framework for Agents' Setup and Organization

In this section, we introduce and analyze a framework for an automated startup and configuration mechanism for a hierarchical structure of the distributed agents in the system. The startup stage is essential to guarantee the accurate and efficient operation of the middleware infrastructure and the applications using it. The main goal here is to provide system startup and configuration with minimum user involvement. A number of issues such as how the agents are connected and how they view one another are considered. The mechanism for adding and removing agents from the system and their effect on the configuration are also studied. Three protocols are introduced for automatic startup, leader recovery, and agent update.

5.1 Hierarchical Structure of Agents

The distributed agents in the system need to communicate among themselves to perform their required operations. However, the structure in which these agents are organized has a strong impact on how efficient they operate. Within a single cluster or a limited number of machines participating in the system, a linear structure is sufficient for the agents to communicate and achieve their functionality. However, this requires agents to be fully connected, which may not always be possible. In addition, the linear structure causes considerable delays for some operations that need to be performed on all participating machines. To overcome these limitations, we designed a hierarchical structure where agents have multilevel connections in the system. Generally, a networked heterogeneous system composed of clusters and multiprocessor machines forms the top level of the hierarchy. Within each of these machines or clusters, one or more levels may be formed, depending on the type of machine and number of nodes/processors in it. For example, a cluster of a small number of nodes remains in a single level, but one with a large number of nodes will have multiple levels in a hierarchy. In this structure, agents are in one of two modes of operation:

1. *Leader agent* (called *leader* hereafter): An agent that manages and controls a set of other agents under its control. Leaders at the same level communicate with one another directly.
2. *Regular agent* (called *agent* hereafter): An agent that performs the regular agent operations. Agents under the control of the same leader should be able to communicate with one another and their leader directly. In addition, agents in different layers, but in the same physical cluster (with direct links between the nodes) communicate with one another directly.

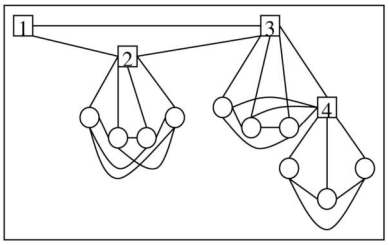

Fig. 2. An example of a hierarchical agent configuration.

Shown in Fig. 2 is an example of a hierarchical structure of a networked heterogeneous system, which we will refer to in the rest of this section. The squares denote leader agents, while the circles represent agents. Some machines such as SMP (symmetric multiprocessing) or MPP (Massively Parallel Processing) machines need a single agent to handle the resources (e.g., leader 1), while others such as clusters need an agent for each node. The connecting lines represent bidirectional communication links between nodes. However, at the top level of the hierarchy, the links between leaders (e.g., 1, 2, and 3) may be a multihop path. The protocols introduced are based on the following general assumptions:

1. Nodes within a cluster are fully connected (e.g., nodes under leader 2, 3, or 4), including nodes belonging to different subtrees (e.g., all nodes under leader 3 are fully connected).
2. In some cases, nodes from different clusters (or machines) may not see each other directly (e.g., nodes under leader 2 cannot see nodes under leader 3).
3. All machines in the system must be fully connected in the sense that at least one node from each machine can see at least one node from each of the other machines.

Before getting into the details of the structure and required mechanisms, we define some basic terms that will be used throughout this section.

1. Virtual cluster: A collection of nodes within one large cluster that form one group of agents and their leader (a subtree). A large cluster is divided into multiple virtual clusters to make communications and management more efficient (e.g., the agents led by leader 4 in Fig. 2).
2. Head node: In some cases, a cluster has a single node that is connected to other machines or clusters. This node is called the head node and has a leader agent residing on it.
3. Local node: From the viewpoint of an agent/leader, the node where it resides is its local node.
4. Remote node: From the viewpoint of an agent/leader, nodes other than its local node are remote nodes.

5.2 Agent Control Messages and Operations

The protocols introduced here require a number of standard control messages that the agents use to communicate and exchange information. These messages, referred to as the middleware control messages (MCM), are defined here.

1. *Leader Advertisement Message* (LAM): A broadcast message sent by a newly created leader to inform other existing leaders of its birth. LAM contains the leader's ID (a unique identifier acquired at startup) and its address information.
2. *Agent Monitor* (AM): Periodic messages sent by leaders to one another and to descendant agents to check if they still exist.
3. *Leader Advertisement Acknowledgment Message* (LAAM): Sent by a leader upon receiving a LAM or AM from another leader. LAAM contains respondent's ID and address information.
4. *Agent Activation Message* (AAM): Sent by a leader to activate descendant agents. It contains the leader's ID and address information, in addition to an activation command.
5. *Agent Monitor Acknowledgement* (AMA): Sent by an agent in response to an AM or AAM. It contains the sender's ID, address, and resources information.
6. *Leader Not Responding Message* (LNRM): Sent by a leader that does not receive an LAAM from another leader in response to the AM message, to all leaders at the same level, and the leader's parent if one exists. It contains the sender's ID and the non-responding leader's ID.

For the agents to operate efficiently, they need a startup protocol to automatically identify and communicate with one another. The initial stage requires manual installation of the first leader agents on the head nodes. The leaders then start the startup and automatic configuration phase.

1. Each leader is responsible for performing the following tasks:
   a. Execute the startup protocol to automatically acquire connectivity and operational information in the system.
   b. Periodically perform availability checks of the leaders and descendant agents. If a leader or agent does not respond, activate leader recovery or agent update protocols.
   c. Perform object routing for other agents to ensure full connectivity with other clusters and machines in the system. Many routing protocols can be adapted for this system, but the discussion of the routing details is beyond the scope of this paper. One suitable example is the content-based object routing technique called Java Object Router (JOR) [24].
2. Each agent should
   a. on activation (by receiving an AAM), find and register local node resources information. Resources include available CPUs, CPU power, storage and memory capacity, etc.,
   b. respond to the leader with an AMA message containing the agent's ID, address, and resources information, and
   c. receive and locally update the neighbors' addresses from the leader for future interprocess communication.

When information becomes available, agents and leaders communicate through the created hierarchical structure where agents collaborate to satisfy user job requirements efficiently. The periodic availability checks can be fine-toned to the system properties to minimize the number of checks performed. This mainly relies on the stability of the system used. If the system is stable and has low probability of failures, then the period between checks can be set to be long, thus reducing the total messages exchanged. However, if the system includes unreliable components or is connected through unreliable communications links, the period should be short enough to discover failures and recover quickly to minimize job failures.

5.3 Leader Startup Protocol

This protocol is designed to assist in automating the startup and configuration of leader agents. The outcome of this protocol is to have leaders acquire full resource information about their descendant agents (including virtual clusters) and routing information about other leaders. In addition, all agents within the same cluster (or virtual cluster) need to have address information of their leader and that of one another. Another important aspect of this protocol is that it allows agents and leaders to be easily added to the system with minimum user intervention. The protocol works as follows:

1. The new leader, L_x, constructs an LAM with its information and broadcasts it on the network.
2. On receiving LAM from L_x, a leader registers received ID and address and sends LAAM to L_x.
3. On receiving the LAAM, L_x updates the address and routing information.
4. L_x initializes the resource table to its local node's available resources and remotely starts accessible agents on the cluster or networked system by broadcasting an AAM.
5. On receiving the AAM from L_x, an agent starts up, constructs an AMA, and sends it to L_x.
6. On receiving an AMA from an agent, L_x updates the resources and routing information.
7. If the number of agents activated is higher than a preset threshold, L_x activates one of the agents to be a leader and assigns some of the agents as its descendants. The new leader performs all the leader operations for the agents under its control.
8. Step 7 is repeated as necessary to evenly distribute agents and leaders to form a balanced hierarchical structure of agents.

The success of this and other protocols and the proper functionality of the agents rely on having a suitable naming (identification) scheme for the agents. Many mechanisms can be used; however, for the system to be scalable, the naming scheme needs to be scalable also. One suitable scheme is the hierarchical naming used for the Internet. Here, the leaders on the top level of the hierarchy take a common root name followed by each machine/cluster name. The next levels use their leader's name as a prefix to their names. For example, assume that the structure shown in Fig. 2 belongs to UNL and, then, the top-level leaders can be UNL.L1, UNL.L2, and UNL.L3. Leader 4 is then called UNL.L3.L4 and the agents under leader 2, for

example, are called UNL.L2.A1, UNL.L2.A2, etc. Such a scheme, while potentially complicated for a small system, allows the system to systematically grow without any need to change previously assigned names or the naming scheme itself. This also allows agents to use the machines' actual Internet URLs as their names, thus allowing easy access through the Internet. Adopting this scheme, however, requires some form of neighbor discovery mechanism as in IPv6 [14] to ensure the use of unique names for the participating agents. In general, the overhead incurred in constructing a hierarchical structure is relatively high, thus it may not benefit a system with a small number of nodes. However, it is essential in two environments:

1. The system is composed of multiple smaller systems (clusters, NOW, multiprocessor machines, etc.) that do not have full connectivity to all their nodes. Thus, the head node in each subsystem is assigned a leader that is responsible of connecting it to other subsystems.
2. The system includes very large clusters comprising tens/hundreds of nodes, thus accessing all nodes in a linear fashion is very time consuming. Here, the threshold needs to be selected to optimize the utilization of the suitable structure. Analytical models or experimental evaluations can be used to select that value.

5.4 Leader Recovery Protocol

This protocol is used in case a leader fails to respond to an AM message sent by another leader. If a leader L_x at one level times out before receiving an AMA response from another leader, say L_y, the following steps are taken by L_x to try to recover from the problem.

1. L_x broadcasts the problem to all other leaders at the same level using the LNRM and informs them that it will try to solve the problem.
2. L_x pings the node/machine where L_y resides to see if it is connected and up.
3. If the node is still up, then
   a. L_x initiates a remote agent activation command to reactivate the agent using the AAM and
   b. when the new agent is up, L_x activates it as a leader and sends it all relevant leader information. The new leader, L_y, uses the startup protocol to restore its information.
4. If the agent does not reinitialize (e.g., has been deleted from the node) or the node does not respond (e.g., has been powered off), then
   a. if a connection exists to another node in the cluster, L_x activates that node's agent as a leader. The new leader then assumes its new role and updates its routing and resource information using the startup protocol,
   b. if no connection exists, L_x reports the problem to the administrator and excludes all routing information to the cluster led by L_y from the routing tables. In addition, L_x informs all other leaders of the changes to avoid the failed node.

Within the affected cluster, jobs that do not involve the failed node continue normally. However, jobs involving the failed node will fail unless they utilize their own fault tolerance mechanisms. The protocol's distributed nature makes it possible for more than one leader to try to restore the same failed leader. However, due to the asynchronous execution of the recovery protocol, the probability of multiple leaders simultaneously initiating the leader recovery protocol is very low. In addition, a back off mechanism can be devised so that a leader can decide whether to continue the protocol or stop because another leader has already started it. One possible approach is to use the leader ID such that the leader with the higher ID proceeds with the leader recovery protocol, while others stop. However, in case more than one leader starts the protocol at the same time, an active agent ignores new activation messages, thus will not be affected by the duplication. Moreover, the leaders will eventually receive the broadcast LNRM and respond to it, resulting in all but one leader to terminate the leader recovery protocol.

5.5 Agent Update Protocol

This protocol is used to report changes in the available resources within a cluster or virtual cluster. The protocol is triggered if one or more nodes (other than the head node) in the cluster fail. When a leader L_x does not receive a response from a descendant agent, then L_x

1. Pings the node of that agent to see if it is up and running and still connected to the network.
2. If it is up, then L_x tries to remotely reactivate the agent on that node using the AAM.
3. If the node does not respond, then L_x
   a. reports the problem to the administrator,
   b. excludes the node from the cluster or virtual cluster,
   c. updates the leader's resources information, and
   d. informs all other nodes on the cluster or virtual cluster of the changes.
4. If the node is restored later, the agent on that node informs the leader of its recovery and updates the cluster and itself with the local routing information.

Jobs using the failed node will fail unless they utilize their own fault tolerance mechanisms. In addition, the protocol provides an automatic mechanism for new or recovered nodes to be included back in the system. In this case, the highest incurred cost comes from activating the agent and updating the clusters information. Nevertheless, this only occurs once per reactivated agent. In addition, the overhead here is limited to the cluster or virtual cluster to which the failed node belongs, thus it does not have any effects on the rest of the system.

5.6 Agent Operations

With the hierarchical structure, the operations of the agents are more organized and efficient compared to having a linear structure where all nodes must see all other nodes at all times. The hierarchical structure also utilizes automatic startup and configuration mechanisms and dynamic agent allocations that reduce user involvement. Although the linear mode of operation is efficient with small clusters because no overhead is imposed from the structure, with

the hierarchical structure on large clusters, most agent operations are performed in parallel resulting in faster response times. The hierarchical structure also provides other advantages such as:

1. providing scalable mechanisms to easily expand the system,
2. providing the update and recovery mechanisms for automatic detection of agent failures or change of status/resources and techniques to report errors and adapt to changes,
3. providing routing capabilities in the leaders to facilitate process communications across multiple platforms over multihop links, and
4. making the agents management and monitoring operations more efficient and less dependant on the full connectivity of the system.

6 An Example of Using the Middleware Infrastructure

The middleware infrastructure is capable of supporting different parallel programming models. An example of this support is the implementation of the Java Object-Passing Interface (JOPI) [23]. In addition, distributed applications utilize this middleware infrastructure to facilitate their operation. In this section, we discuss JOPI, which provides APIs similar to MPI and facilitates information exchange using objects. It utilizes the features provided by the middleware, including the scheduling mechanisms, remote deployment and execution of user classes, control of user threads and available resources, and the security mechanisms. In addition, JOPI was designed such that processes communicate directly with one another if all job threads are directly connected. Otherwise, the threads utilize the agents' routing capabilities.

6.1 JOPI Applications Performance

Benchmark programs were written to evaluate the performance of the system using JOPI. All experiments, unless otherwise mentioned, were conducted on Sandhills, a cluster of 24 Dual 1.2 GHz AMD-processor nodes, 256 KB cache per processor, and 1 GB RAM per node. The cluster is connected via a 100 Mbps Ethernet. For these experiments, standard JVM sdk 1.3.1 was used.

6.1.1 The Agent Overhead

To test the agent overhead, Java programs were executed independently (without the agent) and then through the agent. The average execution time for both executions were measured and compared. Currently, a small overhead (around 0.37 percent) was found since the agent is very lightweight. We assume that adding more functions to the agent may introduce additional, but relatively minor delays. In addition, the overhead is relatively independent from the application, thus it will increase only as the number of processors or machines used increases. The main reason for this is that parallel applications utilize the middleware to deploy and start execution, but then they execute independently from the agents except in few special cases.

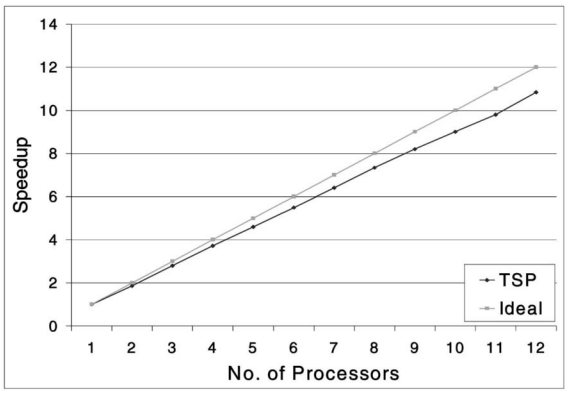

Fig. 3. Speedup results for TSP (22 cities).

6.1.2 Traveling Salesman Problem (TSP)

The algorithm is based on branch-and-bound search [21]. This problem required using many of JOPI's primitives to implement an efficient load-balanced solution. Broadcast was used to distribute the original problem object to processes and to broadcast the minimum tour value found, thus allowing other processes to update their minimum value to speedup their search. Asynchronous communication is used by processes to overlap the reporting of their results to the master with other tasks. The results, as shown in Fig. 3, show good speedup with growing number of processors and fixed problem size.

6.1.3 Experiments on Heterogeneous Platforms

These experiments show the capabilities of the middleware to support the execution of parallel applications on heterogeneous platforms with minimum user involvement. All experiments used standard JVM sdk 1.3.1 on combinations of the following platforms:

- CSNT: 3 CPUs, Intel x86 700MHz, 1.5GB RAM, OS: Windows 2000 advanced server.
- RCF: SGI Origin 2000, 32 processors, 250 MHz, 4MB cache, 8GB RAM, OS: IRIX 6.5.13.
- Sandhills: Cluster, 24 nodes, dual 1.2 MHz AthlonMP, 256KB cache, 1GB RAM, OS: Linux.

To fairly compare the performance, the sequential running time for the program was measured on each platform. Speedup is calculated with respect to the fastest sequential time in the configuration used. A more formal model to calculate the performance of parallel applications on heterogeneous systems can be found in [3].

Matrix Multiplication (MM). A dense matrix multiplication (MM) algorithm [17] is used with load balancing mechanism and synchronous point-to-point communication. A matrix of size $1,800 \times 1,800$ floating numbers was used, with a stripe size of 300 rows or columns. The results on Sandhills, RCF, and CSNT are listed in Table 1.

Traveling Salesman Problem (TSP). The algorithm used is the same as in Section 6.1.2, using the machines CSNT and Sandhills. TSP was executed for 22 cities using different configurations of heterogeneous processors from Sandhills and CSNT (see Table 2).

TABLE 1
Performance Measurement for MM on a Heterogeneous System

Matrix Multiplication (MM)					
No. Processors				Elapsed Time	Speedup
Total	SH	RCF	CSNT		
1	1	0	0	280.83	1.000
1	0	1	0	807.65	1.000
1	0	0	1	265.19	1.000
3	3	0	0	100.93	2.782
3	0	3	0	416.49	1.939
3	0	0	3	100.24	2.646
6	3	0	3	55.708	4.760
12	6	3	3	42.992	6.168
16	13	0	3	34.591	7.666

TABLE 2
An Example of a Hierarchical Agent Configuration

Traveling Salesman Problem (TSP)				
No. Processors			Elapsed Time	Speedup
Total	SH	CSNT		
1	1	0	35609729	1.000
1	0	1	37839885	1.000
2	0	2	18765547	2.016
2	2	0	18068563	1.971
4	2	2	14112931	2.523
6	4	2	7782309	4.576
8	6	2	5633393	6.321

6.2 Discussion

The infrastructure provides a platform for parallel Java using JOPI, which achieves good performance. However, JOPI, in its current form, is most suitable for applications that have high computation to communication ratio or coarse grain parallelism, but can be optimized to handle finer grain parallelism. In addition, the varying specifications of the processors used indicate the possibility of achieving more speedup and faster response times by distributing tasks based on their suitability to the platform. For example, if some tasks require excessive data sharing, they can be assigned to a multiprocessor parallel machine, while relatively independent tasks can be assigned to a cluster. More detailed experiment results and comparisons with MPI can be found in [23]. As described earlier, JOPI utilizes the middleware infrastructure; however, the experiments show that the agents impose a very small overhead while providing efficient and flexible functions for JOPI. In addition, the communication overhead incurred by the agent occurs mostly with the initial deployment of the application, which then relies on the programming model's implementation of the interprocess communication functions. The agents also allow user jobs to be deployed and executed on remote machines transparently, requiring no user involvement other than specifying the number of processors needed. This, in addition to Java's portability, has allowed easy utilization of multiple distributed platforms of different specifications to execute a single parallel application.

7 Conclusion and Future Work

The middleware infrastructure provides services to support the development of high-performance parallel and distributed Java applications on clusters and heterogeneous systems. The distributed agents collectively form the middleware infrastructure that supports different parallel programming models, in addition to distributed applications. The middleware provides APIs that enable programming models developers to build different parallel programming models and tools. In addition, the middleware allows the distributed/parallel application developers to build, deploy, monitor, and control their applications, which can be written using the middleware directly or the programming models provided on top of it. Some of the main advantages of using distributed agents and the hierarchical structure are:

1. Portability: The system is fully portable, allowing it to support seamless execution of parallel applications across different multiple platforms. Here, the agents distribute user processes to remote machines, deploy them remotely as threads, monitor their progress, and allow users to manage and control their applications.
2. Expandability: The hierarchical structure allows easy additions/removals of agents from the system transparently from the programming models and applications.
3. Flexibility: It is easy to modify or replace the systems components such as the scheduler and deployment mechanisms, and add more features such as fault tolerance and resource discovery without requiring changes to the applications or the programming models. The programming model implemented using the middleware is also free to utilize some or all of the functions provided by the middleware, while utilizing its own specialized functions as well.
4. Security: The agent's security module and its support for different execution modes provide the user applications with different levels of access to the machines used. Thus, limiting the ability of malicious processes to damage or disrupt the system resources (e.g., manipulating files, executing restricted system calls, etc.) or other users jobs. In addition, this module can be further enhanced to support policy driven security measures defined by the users/administrators and implemented at the middleware level to enforce the required authentication, authorization, and access controls.
5. Resource Management: Agents, collectively, have information about all the resources, which provides a distributed information base of system resources. Thus, they can collaborate to provide efficient and comprehensive resource discovery and management tools.

An additional unique feature in the system is the preservation of the compatibility with available JVMs and the layered approach that separates the programming models from the runtime support services. Furthermore, the framework for a hierarchical organization of agents

provides efficient agent communications and operations, while the startup mechanisms automate the configuration of agents and allow easy expansions.

The experiments conducted show that the system performs well. However, there are numerous opportunities for enhancing the utilization of resources and quality of service for high-performance Java through the cooperation and coordination among agents. These could be dynamic resource discovery and recovery modules, collaborative garbage collection, and sophisticated dynamic scheduling mechanisms for user threads. Additional functions can also be integrated to the infrastructure such as fault tolerance and dynamic load balancing. In addition, we are working on developing a distributed shared Java object space based on this infrastructure.

ACKNOWLEDGMENTS

This project was partially supported by a US National Science Foundation grant (EPS-0091900), a Nebraska University Foundation grant (26-0511-0019), and a University of Nebraska Academic Priority grant. The authors would like to thank the members of the Secure Distributed Information (SDI) group and the Research Computing Facility (RCF) at the University of Nebraska-Lincoln for their help and support. They would also like to extend their gratitude to the reviewers of this special section for their invaluable comments and suggestions that helped improve the quality and organization of this paper. A preliminary version of portions of this paper was presented at IEEE Cluster 2002 [1].

REFERENCES

[1] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson, "An Agent-Based Infrastructure for Parallel Java on Heterogeneous Clusters," *Proc. Fourth IEEE Int'l Conf. Cluster Computing,* pp. 19-27, 2002.

[2] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson, "A Comparative Study of Parallel and Distributed Java Projects for Heterogeneous Systems," *Proc. IEEE IPDPS, Workshop Java for Parallel and Distributed Computing,* 2002.

[3] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson, "Modeling Parallel Applications Performance on Heterogeneous Systems," *Proc. IEEE IPDPS, Workshop Advances in Parallel and Distributed Computational Models,* 2003.

[4] Y. Aridor, M. Factor, and A. Teperman, "Transparently Obtaining Scalability for Java Applications on a Cluster," *J. Parallel and Distributed Computing special issue—Java on Clusters,* vol. 60, no. 10, pp. 1159-1193, Oct. 2000.

[5] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F.C. Eigler, and G.R. Gao, "ABC++—Concurrency by Inheritance in C++," *IBM Systems J.,* vol. 34, pp. 120-137, 1995.

[6] A. Bik and D. Gannon, "JAVAR: A Prototype Java Restructuring Tool," http://www.extreme.indiana.edu/~ajcbik/JAVAR/index.html, July 2003.

[7] A. Bik and D. Gannon, "JAVAB: A Prototype Bytecode Parallelization Tool," http://www.extreme.indiana.edu/~ajcbik/JAVAB/index.html, July 2003.

[8] T. Brecht, H. Sandhu, M. Shan, and J. Talbot, "ParaWeb: Towards World-Wide Supercomputing," *Proc. Seventh ACM SIGOPS European Workshop,* pp. 181-188, 1996.

[9] *High Performance Cluster Computing: Architectures and Systems,* R. Buyya, ed., Prentice Hall Inc., 1999.

[10] M. Campione, K. Walrath, and A. Huml, *The Java Tutorial Continued: The Rest of the JDK.* The Java Series, Addison-Wesley Publication Co., 1998.

[11] D. Caromel, W. Klauser, and J. Vayssiere, "Towards Seamless Computing and Metacomputing in Java," *Concurrency: Practice and Experience,* vol. 10, pp. 1043-1061, 1998.

[12] B. Carpenter, G. Fox, H.K. Lee, and S.B. Lim, "Translation Schemes for the HPJava Parallel Programming Language," *Proc. 14th Int'l Workshop Languages and Compilers for Parallel Computing,* http://www.hpjava.org/, Aug. 2001.

[13] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox, "MPI for Java, Position Document and Draft Specification," Technical Report TGF-TR-03, Java Grande Forum, http://www.javagrande.org/jgpapers.html, July 2003.

[14] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 1883, http://www.faqs.org/rfcs/rfc1883.html, July 2003.

[15] K. Edwards, *Core Jini,* second ed., Prentice Hall PTR, Dec. 2000.

[16] A.J. Ferrari, "JPVM: Network Parallel Computing in Java," Technical Report CS-97-29, Dept. of Computer Science, Univ. of Virginia, http://www.cs.virginia.edu/~ajf2j/jpvm.html, July 2003.

[17] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn, "Analysis of a Class of Parallel Matrix Multiplication Algorithms," *Proc. Int'l Parallel Processing Symp.,* 1998.

[18] *Software Agents for Future Communication Systems,* A. Hayzelden and J. Bigham, eds., Berlin Heidelberg: Springer-Verlag, 1999.

[19] M. Izatt, "Babylon: A Java-Based Distributed Object Environment," Master's thesis, Dept. of Computer Science, York Univ., Canada, July 2000.

[20] M. Izatt, T. Brecht, and P. Chan, "Ajents: Towards an Environment for Parallel Distributed and Mobile Java Applications," *Proc. ACM Java Grande Conf.,* pp. 15-24, 1999.

[21] R. Karp and Y. Zhang, "Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation," *J. ACM,* vol. 40, no. 3, pp. 765-789, July 1993.

[22] V. Laxmikant and S. Krishman, "Charm++: A Portable Concurrent Object Oriented Systems Based on C++," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices,* vol. 28, no. 10, pp. 91-108, Oct. 1993.

[23] N. Mohamed, J. Al-Jaroodi, H. Jiang, and D. Swanson, "JOPI: A Java Object-Passing Interface," *Proc. ACM Joint Java Grande-ISCOPE Conf.,* pp. 37-45, 2002.

[24] N. Mohamed, A. Davis, X. Liu, and B. Ramamurthy, "JOR: A Java Object Router," *Proc. 14th IASTED Int'l Conf. Parallel and Distributed Computing and Systems,* pp. 630-635, 2002.

[25] Myrinet/Myricom Web page, http://www.myrinet.com/, July 2003.

[26] M. Philippsen and M. Zenger, "JavaParty—Transparent Remote Objects in Java," *Concurrency: Practice and Experience,* vol. 9, no. 11, pp. 1125-1242, Nov. 1997.

[27] RMI Web pages at Sun Microsystems, http://java.sun.com/products/jdk/rmi/, July 2003.

[28] J. Squyres, J. Willock, B. McCandless, and P. Rijks, "Object Oriented MPI (OOMPI): A C++ Class Library for MPI," *Proc. Parallel Object-Oriented Methods and Applications Conf.,* 1996.

[29] D. Walker and J. Dongarra, "MPI: A Standard Message Passing Interface," *Supercomputer,* vol. 12, no. 1, pp. 56-68, Jan. 1996.

[30] K.A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P.N. Hilfinger, S.L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java Dialect," *Concurrency: Practice and Experience,* vol. 10, nos. 11-13, Sept.-Nov. 1998.

Jameela Al-Jaroodi received the BSc degree in computer science from The University of Bahrain, Bahrain, in 1993, and the MSc degree in computer science from Western Michigan University, Kalamazoo, Michigan, in 1998. She is currently a PhD candidate in the Department of Computer Science and Engineering, University of Nebraska-Lincoln, Nebraska. Her research interests include distributed system middleware, heterogeneous systems, parallel and distributed programming models, programming languages, wireless networks, and computer security. She has published more than 20 publications in major journals and international conferences. Ms. Al-Jaroodi is a student member of the IEEE, IEEE Computer Society, and IEEE Communications Society.

Nader Mohamed received the BSc degree in electrical engineering from The University of Bahrain, Bahrain, in 1992, and the MSc degree in Computer Science from Western Michigan University, Kalamazoo, Michigan, in 1998. He is currently a PhD candidate in the Department of Computer Science and Engineering, University of Nebraska-Lincoln, Nebraska. His research interests include middleware, networks, cluster and grid computing, and parallel and distributed software. He has published more than 20 technical papers in these areas. Mr. Mohamed is a student member of IEEE and IEEE Computer Society.

Hong Jiang received the BSc degree in computer engineering in 1982 from Huazhong University of Science and Technology, Wuhan, China, the MASc degree in computer engineering in 1987 from the University of Toronto, Toronto, Canada, and the PhD degree in computer science in 1991 from the Texas A&M University, College Station, Texas. Since August 1991, he has been at the University of Nebraska-Lincoln, Lincoln, Nebraska, where he is an associate professor and vice chair in the Department of Computer Science and Engineering. His present research interests are computer architecture, parallel/distributed computing, computer storage systems and parallel I/O, performance evaluation, middleware, networking, and computational engineering. He has published more than 60 publications in major journals and international conferences in these areas and his research has been supported by the US National Science Foundation, DoD, and the state of Nebraska. Dr. Jiang is a member of the ACM, the IEEE Computer Society, and the ACM SIGARCH and ACM SIGCOMM.

David Swanson received the PhD degree in physical (computational) chemistry at the University of Nebraska-Lincoln (UNL) in 1995, after which he worked as a US National Science Foundation-NATO postdoctoral fellow at the Technical University of Wroclaw, Poland, in 1996 and, subsequently, as a National Research Council Research Associate at the Naval Research Laboratory in Washington, DC, from 1997-1998. In early 1999, he returned to UNL, where he has coordinated the Research Computing Facility and currently serves as an assistant research professor in the Department of Computer Science and Engineering. The Office of Naval Research, the National Science Foundation, and the state of Nebraska have supported his research in areas such as large-scale parallel simulation and distributed systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.