

AdJava – Automatic Distribution of Java Applications

Mohammad M. Fuad and Michael J. Oudshoorn

Department of Computer Science

University of Adelaide

Adelaide 5005, South Australia

{Fuad, Michael}@cs.adelaide.edu.au

Abstract

The majority of the world's computing resources remains idle most of the time. By using this resource pool, an individual computation may be completed in a fraction of time required to run the same computation on a single machine. However, distributing a program over a number of machines proves to be a tedious and difficult job. This paper introduces a system, called AdJava, which harnesses the computing power of these under-utilized heterogeneous computers by automatically distributing the user application across the available resources. In addition to providing transparent automatic distribution, AdJava provides load balancing and migration of distributed objects through the use of intelligent software agents. The system provides all this support without any programmer involvement and without modifying the Java Virtual Machine (JVM). AdJava's range of features, combined with its ease of use, makes it a powerful system for distributed computing.

Keywords: Distributed programming, software agents.

1 Introduction

In the past few years, there have been substantial changes in computing practice mainly as a result of the proliferation of low priced powerful workstations connected by high-speed networks. However studies by Tandary *et al* (1996), have shown that many of these workstations are idle or lightly loaded most of the time. If these under utilized resources could be effectively harnessed, the resultant computational power would be equivalent to that of a supercomputer. This paper describes a system that exploits the computing power of these under-utilized heterogeneous computers by automatically distributing a user application across the available resources. This system is significantly different from most other research projects in the way that it tries to hide, as much as possible, the underlying distribution mechanisms and associated issues from the application programmer. This frees the application programmer to focus on the complexities of the application on hand, rather than being distracted by the issues associated with distribution.

Most other research projects either attempt to improve the underlying communication mechanism (Detmold *et al* 1999, Falkner *et al* 1999, Oudshoorn *et al* 2000) or only try to implement migration (Sekiguchi 1999) or take a completely different approach, such as using distributed shared memory (Yu and Cox 1997), in providing support for distributed object programming. Those systems either generate additional burdens for the programmer (in the form of workload in understanding the underlying system) or do little to hide the distribution process. Few, if any, of these research projects focus on automating the distribution process. This paper presents a system that not only automates the distribution process but also automatically provides several other features such as migration and load balancing through the use of intelligent software agents.

1.1 Concept and architecture

This paper describes a Java-based (Sun Microsystems 2001) software-agent-oriented distributed computing infrastructure. The system consists of a dynamic pool of "trusted" computers of heterogeneous software and hardware platforms connected by a Local Area Network (LAN) or Wide Area Network (WAN). The system models a parallel application as a collection of distributed objects running concurrently on different computers. On each of these remote computers, a multi-threaded software agent handles collaboration between objects in the system. The agent resides on top of the Java Virtual Machine (Meyer and Downing 1997) and interacts with its own host, and agents on other hosts. Figure 1 shows an example of distributed computation using the AdJava system. The objects in the user program are distributed over a number of machines where agents are running. Each agent in the system knows about other agents and can communicate with each other. Agents can intercept any invocation to an object. The user puts special keywords into the program to distinguish distributed objects from non-distributed objects. Then, the corresponding distributed objects are created to execute each sub-task concurrently at different machines. Making use of the automatically generated RMI *stub* and *skeleton* (Sun Microsystems) for each distributed object, users can program the distributed objects in the same way as standard objects. AdJava supports the *peer-to-peer* paradigm along with the traditional *client-server* paradigm. In general, each agent in the system can act as both a server and a client of different distributed objects. Communications between the distributed objects are permitted, but such interactions should be kept to a

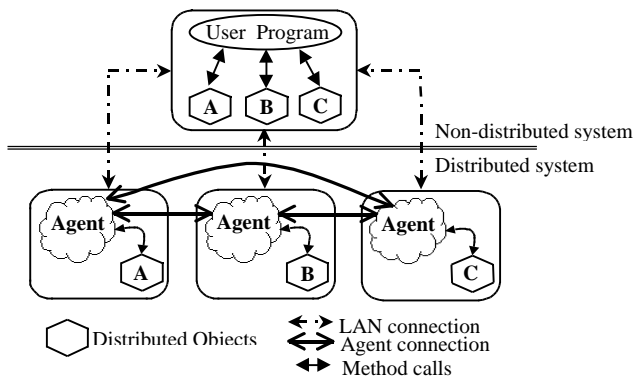


Figure 1: An example of distributed computing using AdJava.

minimum in order to reduce the inter-node communication overheads.

1.2 Goals

The most important goal of the AdJava system is to provide the user with a simple but robust infrastructure to automatically distribute applications across the available resources. The system alleviates the complexities and problems associated with manual distribution and provides better throughput and execution time by managing all of the distribution issues through the use of autonomous software agents. AdJava allows users to seamlessly create objects on different machines to access these objects in a fashion similar to local objects. Distributed objects are able to produce output to files and consoles in a similar manner to that of a local object. The infrastructure is also simple to use. It operates by establishing agents on remote machines and running a modified user program on each machine. The system provides transparent automated distribution of applications across a network and, in situations of heavy load, the agents migrate remote objects as necessary. AdJava provides all this support without any programmer involvement.

While Java provides an excellent base for creating applications, the features available for distributed programming are difficult to use. The aim of this project is to provide access to existing features and some additional new features to user programs without the need for any user involvement. AdJava provides powerful support for distributed objects through the use of software agents, while simplifying the programmer's task and maintaining 100% Java compatibility. Several systems exist which deliver varying degree of support for distributed systems. These systems are examined in Section 2. However, none are as ambitious as AdJava in terms of the support provided to the programmer.

1.3 Scope

The purpose of this work is to provide a simple yet efficient infrastructure that supports transparent automatic distribution of user program. There are several important issues that must be provided to make a distributed system successful (Alexandrov *et al* 1997). These issues include:

- **Automatic distribution:** Objects are distributed across the network transparently and without any user involvement.
- **Ease of use:** Ease of use is the key property of AdJava. The user only needs to add a single keyword to make objects distributable.
- **Scheduling and Load balancing:** Through the use of software agents, scheduling of distributed objects is carried out dynamically. Load balancing of remote objects by migrating active objects to another lightly loaded machine in the system is handled by the software agents.

1.4 Contributions and outline

Converting a multi-threaded Java program into a distributed Java application is not an easy task. A huge gap exists between a multi-threaded and a distributed Java application that forbids simple code transformation in order to build distributed applications from multi-threaded applications. Java and Java/RMI place a heavy burden on the programmer because they require deep modification of existing code in order to convert local objects into remotely accessible ones. AdJava provides a system that makes it easy for programmers to convert a multi-threaded parallel program into distributable one. AdJava transforms local object into distributed objects without user involvement and handles issues related to distributed Java objects. The programmer does not need to worry about the distribution of the resulting program; AdJava deals with everything related to distribution on behalf of the programmer.

The use of remote method invocations often limits interaction with remote objects. Traditionally, remote objects only generate output on the host upon which they are executing. AdJava provides support for remote input from files and output to console and files. This is done in a manner which is easy to use and operates transparently even when objects are migrated.

AdJava contributes new features and approaches to the area of distributed object computing in Java. However, the key contribution of AdJava is the integration of both new and previously researched (Izatt *et al* 1999, Christiansen *et al* 1997, Philippsen and Zenger 1997) features, into a single comprehensive system, which is easy to use. While other systems implement some of the same functionality in different ways, none combine the breadth of features with the ease of use of AdJava.

This paper is organized as follows. Section 2 looks into the field of Java-based distributed processing. In Section 3, we will look into the different aspects of our system. Section 4 presents measurements from our system. These measurements shed light on those aspects of the system that are beneficial to distributed application. Section 5 concludes the paper.

2 Background and related work

Considerable research has been done in the field of Java based parallel and distributed programming. There are several systems (Izatt and Chan 1999, Christiansen *et al*

1997, Philippsen and Zenger 1997) that have been developed which leverage the Java Virtual Machine to provide distributed or parallel computing environment on heterogeneous computing platforms.

JavaParty (Philippsen and Zenger 1997) is the closest research project in terms of matching our goals. JavaParty uses a *remote* class modifier to denote a class that should be used in a distributed fashion. One of the major drawbacks of JavaParty is that it passes non remote objects in parameters by copy and not by reference. So, if a non-remote object refers to a structure of non-remote objects, then the whole graph of objects is copied across to the recipient. To overcome this problem, AdJava uses RMI callbacks and gives the user the option to pass objects by references or by copy. Another drawback of JavaParty is that it creates ten Java byte-code files for every single distributable class. JavaParty can only migrate those objects that have not started executing. It also does not provide a user friendly interface through which user can interact with the system.

Ajents (Izatt *et al* 1999) utilizes Java RMI for its communication mechanisms. Ajents support implementation of parallel, distributed and mobile Java applications over a heterogeneous computing resource. An Ajent is a mobile agent that can be relocated to respond to load balancing requests but is not autonomous. Ajents is implemented as a collection of Java class files, which makes programming difficult for programmers. Ajents is not transparent to the programmer and programmers need to have a detailed understanding of the underlying system in order to program using Ajents; this sacrifices simplicity. Ajents require programmers to keep track of which objects are local and which are remote and to ensure that the proper interface is used for remote objects. In AdJava, users do not need to worry about local and remote objects or any interfaces, underlying agents keep tracks all of these and provides user a totally transparent system. Another major drawback of Ajents RMI interface is that the method and parameters passed as parameters to the RMI call can not be checked at compile time, nor can the types of the parameters that are to be passed to the specified method. As a result it is not possible to detect errors that might otherwise be detected at compile time. In AdJava, all programming errors are detected at compile time and the system aborts if any compile time error appears.

Javelin (Christiansen *et al* 1997) provides an infrastructure for global computing. Javelin allows a machine connected to the Internet to make a portion of their idle resources available to remote clients, while at other times utilizing resources from other machines when more computational power is needed. Due to the reliance upon applets, Javelin does not support object interactivity or advanced features such as migration.

Charlotte (Baratloo *et al* 1996) and Java/DSM (Yu and Cox 1997) provide support for distributed shared memory applications. Although Charlotte is an interesting approach to distributed programming, the variable access methods add extra programming complexity to gain access to shared variables, in addition to the overhead required to keep memory consistent across nodes. In

contrast, adJava provides distributed object access using remote method invocation rather than shared memory. In DSM, the authors make changes to the JVM that restricts its use across heterogeneous network.

Most of the above systems can only provide output to the host upon which the remote objects are executing. AdJava includes built in support for remote input and output to and from consoles and files.

ARMI (Raje *et al* 1997), RRMI (Thiruvathukal *et al* 1998) and Manta (Maassen *et al* 1999) systems try to alleviate various drawbacks in RMI and provide new RMI style systems with added functionality.

Mole (Straber *et al* 1996) and Aglets (Lange and Oshima 1998) are designed to support Java-based autonomous software agents. These systems concentrate on the independent movement of objects, while providing less support for interaction and control by the object's creator. Along with providing transparent control of objects, agents in AdJava are designed to provide flexible operation and easy interaction with the underlying system.

In case of object migration, Fünfroeken (1998) presents a technique that allows Java programs to migrate both object state and a artificial program counter transparently. This is done by pre-processing the Java source code. Code is added that saves the runtime state at certain points, making it possible to restore this state at the same point. Sekiguchi *et al* (1991) uses a slightly different approach. Instead of saving program state, they use Java's Exception mechanism and special state variables to perform migration. There are several systems (Bouchenak 1999, Peine and Stolpmann 1997) that modify the Java Virtual Machine for migration. This sacrifices the heterogeneous property of Java and restricts portability. AdJava's implementation of transparent migration bears some similarities with Fünfroeken's approach, where users need to explicitly specify where the migration should happen by inserting specific keywords. It differs in that, AdJava determines the point of migration dynamically and takes appropriate steps to perform migration.

3 System architecture

AdJava is designed as an agent-oriented system where the user program is preprocessed and the resulting code is compiled by the regular Java compiler to generate the byte code. The rest of the system consists of a dynamic pool of computers of heterogeneous hardware and software platforms, connected by a LAN or WAN. In each of the remote machines, a multi-threaded agent runs on top of the JVM that interacts with other agents in the system. The user puts special keywords into the program code to distinguish the distributable objects. Along with the modified program, the user also provides a list of remote machines on which the agents are running and where objects can be distributed. Once the objects are distributed and running on remote machines, the agents keep track of remote objects and perform load balancing when required. From pre-processing to load balancing,

everything is performed automatically and transparently without any user involvement.

3.1 Design issues

AdJava is designed as a distributed object oriented system. Java is selected for its platform independent characteristics. Java RMI provides a clean approach to distributed object programming. Other approaches such as Message Passing Interface (MPI) (Message Passing Interface Forum, 1994) and Remote Procedure Calls (RPC) (Birrel and Nelson, 1984) are rejected for several reasons. First of all, they do not support heterogeneous platforms. Both of these systems expose the underlying hardware differences to the programmer. Neither of these systems provides a local view of the remote object. In both cases, users need to do all the low-level networking in order to perform communication across remote hosts.

The design and implementation of AdJava involved making many decisions and tradeoffs in order to achieve a useful distributed object system. Several of the important issues are as follows:

- **Object model:** AdJava uses a dual object model. There are local and remote (distributed) objects. Distributed objects are the basic units for distribution and concurrency. Users have to explicitly specify which objects are to be distributed by using a special keyword. Distributed objects should be computationally intensive to alleviate the relative communication overheads for message passing over the network and to gain maximum performance improvement over a sequential program. Section 3.2 describes the object model in detail.
- **Communication:** Along with providing synchronous communication, AdJava also provide asynchronous method calls to improve overall performance. AdJava uses the message-passing paradigm to communicate between agents on the system. It uses Java's Serialization feature to transfer objects between agents. Generally, AdJava uses User Datagram Protocol (UDP) sockets for message passing and Transmission Control Protocol (TCP) sockets for object transfer.
- **Location transparency:** AdJava employs a location transparent approach because it provides an easy and user-friendly environment. In AdJava, the syntax for local objects and distributed objects are the same. It is not necessary to consider the location of the distributed objects. The system dynamically selects a appropriate node to share the workload without any user intervention. After migration of objects, AdJava uses an exception mechanism system to update object references across machines.
- **Object migration:** AdJava uses a combination of pre-processing in source code and Serialization for object migration. Once objects are suspended, they are serialized and migrated to a new host, where they are restarted using a unfolding technique which is describe in more detail in Section 3.8.4.

- **Configuration:** AdJava is composed of a special root server and a collection of agents. The root server is the name of that host on which the user is submitting the program for distribution over the network. The root server acts as the initial contact point for all agents and every agent has the address of the root server. Through the root server, all agents receive the initial set-up information about every other agent in the system. Section 3.8.5 provides additional detail on reference updating. In general, all agents can also communicate directly with each other.

3.2 Program structure and pre-processor

Using AdJava, programmers can easily turn a multi-threaded Java program in to a distributed program by identifying those objects that should be spread across the distributed environment. AdJava uses a special keyword *Distribute*, to indicate objects that can be distributed. The new keyword is the only extension AdJava makes to standard Java.

In the following code segment, *aObject* is a multi-threaded object.

```
...
foo aObject = new foo (...);
...
class foo extends Thread
...

```

To make the *aObject* distributed, all that needs to be done is to use the special keyword in the object declaration.

```
...
Distribute foo aObject = new foo (...);
...
class foo extends Thread
...

```

This is the only change needed to make that object distributed. Without the *Distribute* keyword, the same program compiles using the standard Java compiler and runs in a single machine as a concurrent program. Programmers should remember that, in the distributed objects, no static fields and methods may be used. It is the programmers' responsibility to modify the code to reflect these changes. Communications between distributed objects are permitted, but such interaction should be kept to a minimum in order to reduce the inter-agent communication overheads.

Once the programmer adds the *Distribute* keyword to the concurrent program it becomes ready for pre-processing. If the programmer compiles the original program without adding the keyword, the program will only run on the root server.

3.3 The root server

The root server has a well-known fixed address. The root server executes the *root daemon*, which is responsible for the setup and running of the whole system. After pre-processing the user program, the root daemon performs handshaking with the agents on remote machines. This is done for several reasons. First of all to check whether the agents are running where they should be and that they can

access predefined ports. Also, during handshaking, the agents and the root daemon pass information between them. The root daemon passes application specific information, such as the name of the object, number of agents in the system and agent addresses. On the other hand, the agents pass their current load status which is used by the root daemon to distribute objects accordingly. For instance, if an agent indicates that it is lightly loaded, then the root daemon may give it more objects to run, than a heavily loaded machine. Depending on the handshaking information and parameter-passing mode, the root daemon uploads the stubs and associated files into the agents. Once all objects are distributed, the root daemon uses reflection to invoke the *Ignition* method (equivalent to the main method in the user program) to start the remote objects running. The root daemon has two major threads that perform all background work once the system starts running. One keeps track of the root registry and the other keeps track of the remote input and output from remote agents. The root registry is a table storing all the remote object and agent's information. This is used when shutting down the whole system. To shutdown or reset the whole system, the root daemon pass control messages to agent addresses stored in the root registry. By replicating the root registry to other agents, agents can behave as root server and AdJava would have better fault tolerance. After migration of an object, the table is updated to reflect the new address of the object.

3.4 Parameter passing

Since RMI uses object serialization to pass parameters and return values in RMI calls, it can pass the true type of an object from one virtual machine to another in a remote call. In RMI, a non-remote object that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation is passed by copy. So, in this case, the content of the non-remote object is copied before invoking the call on the remote object. When a non-remote object is returned from a remote method invocation, a new object is created in the calling virtual machine. On the other hand, when passing a remote object as a parameter or return value in a method call, it is passed as reference. However, there are situations when non-remote objects are required to be passed by reference. For instance, if a non-remote object is passed as a parameter that itself refers to other non-remote objects, then the whole graph of objects is copied to the recipient. If the receiving method changes the graphs, then there will be different versions of it in the system. As RMI does not support pass by reference for non-remote objects, AdJava employs RMI's callback feature (Sun Microsystems 2001) to simulate pass by reference for non-remote objects.

3.5 Object distribution strategy

AdJava uses a simple distribution policy to distribute objects to available machines. At startup, the root daemon gets load values from all the agents indicating how loaded each node is. Depending on this information and on the number of distributed objects and machines in the system, AdJava distributes objects among the machines. If the

number of objects to be distributed is more than the number of machines in the system AdJava distributes more than one objects to those machines that are loaded lightly compared to other machines in the system. By default, every machine in the system will get at least one object to run. The challenge with distributing objects transparently is to provide unique RMI standard names to objects such that they can be used to bind objects into the RMI registry and to locate an object in the registry. So AdJava uses a simple naming policy which assign each object a unique name as follows:

`rmi://host name:default port/obj n`

So incrementing the default port number and the object number (n), AdJava assigns different objects in the same machine with distinct names. Figure 2 shows AdJava's transparent distribution mechanism. To measure the load in each of the machines, the agent checks how many times the CPU was used in the last one minute. So, for instance in Figure 2, if the root server has six objects, it will distribute that object to **argon** and not to **radon**. This

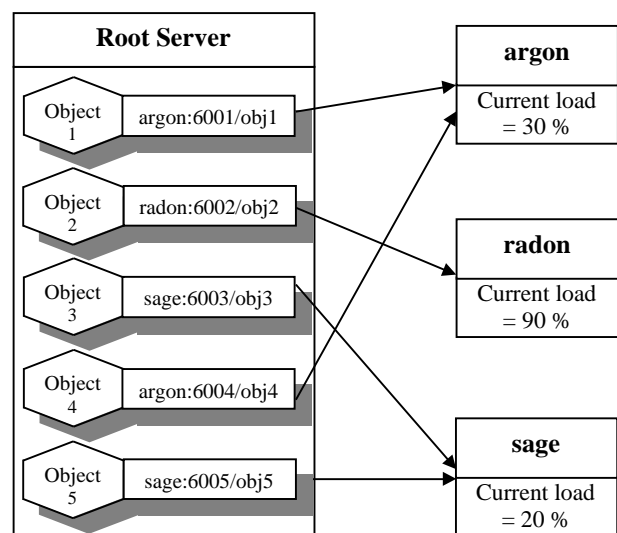


Figure 2: Transparent object distribution in AdJava.

is because **argon** is in the top of the host list and **radon** is busy, i.e. its load exceeds a predefined threshold.

3.6 Remote Input and Output

AdJava includes features designed to allow input and output to be processed by remote objects. From the perspective of the user on the root server, the remote object produces output and requests input as if it were actually executing upon the root server. Two issues must be addressed. It should be easy to identify which remote host was the source of the output. AdJava provides this support by appending the host name to the front of the data packet containing the output. Another issue is the synchronization problem, namely how to synchronize output from several hosts to a single output device. AdJava uses a sequence number and in the root server it buffers the output and produces synchronized output depending on the sequence number.

3.6.1 File I/O

From the user perspective, file access through AdJava occurs in the same manner as in standard Java. During pre-processing, AdJava checks the remote objects for any file I/O and changes the code accordingly. When a file is to be opened remotely, the file is serialized and transferred to that host for input operations. In the case of output to a file, AdJava changes the code to provide remote file output. The pre-processor adds the specific statements for each of the file I/O operations using the **RemoteIO** object. The agents have access to the **RemoteIO** object and when a file is being closed, the subsequent **setFlag** method indicates the **RemoteIO** daemon to transfer the corresponding file to the root server. Different objects may write to a file with the same name in different hosts. When those files are moved to the root server, to stop over writing the same file twice, the root server adds prefixes in the file names to distinguish between files. In providing remote I/O, the challenge is to provide the same transparency after an object is being migrated. If AdJava provides the remote I/O facility as a wrapper class to the remote objects, then migrating an object will invalidate the local file references. As AdJava provides the service as a separate object inside the remote object, when the remote object is migrated, the associated **RemoteIO** object is also moved to the new location. In this way AdJava bypasses problems associated with invalid references. However, as a result of this design decision, AdJava supports only independent file access. That is, if multiple remote objects are trying to access the same file, they may do so, but each must separately open the file and use its own file pointer.

3.6.2 Console I/O

Supporting reads and writes to the console introduces challenges that are different from that of file access because of the atomic nature of the console. While multiple objects can simultaneously open the same file, Java does not allow that behavior with the console. Even if multiple objects hold a reference to the console, they would share the same reference, whereas with remote files, they each could be holding a unique reference. In Java, this is represented by the static variables **System.in**, **System.out** and **System.err**, which provide user programs with access to standard input, output and error respectively.

AdJava preserves this behavior by allowing the access to the console through the **RemoteIO** object. The **RemoteIO** object in the agent and in the root server uses UDP connections to transfer output from agents to the root server. So, although it seems that different objects are using different consoles on different machines, only one console appears on the root server and output is displayed in the sequence as produced by different distributed objects. Figure 3 shows the basic idea behind the AdJava approach. Whenever a **RemoteIO.out** statement is executed, the Remote I/O daemon sends the corresponding parameter as an UDP packet to the root server. Before sending the packet, the Remote I/O daemon adds the sequence number and the agent address

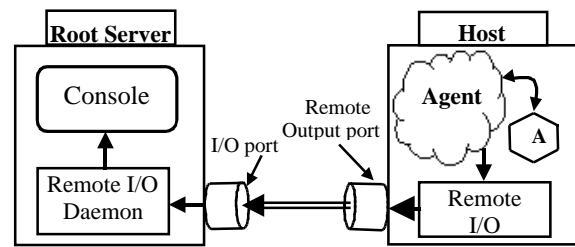


Figure 3: Remote console output.

in front of the packet. On the other side, the root daemon uses this information to produce a synchronized output. Currently AdJava does not support console input from a remote host.

3.7 The agents

The agent is a special software component that has limited autonomy. It behaves like a human agent, working for some client in pursuit of its own agenda (Xu and Wims 2000). In AdJava, agents reside on top of the JVM and interact with the host. The agent is designed as a multi-threaded application where each of the threads has a specific job. The agent always listens to a port for any message from either the root server or from other agents in the system. At startup, it performs handshaking with the root server and dynamically loads object classes.

Once the root server invokes its distributed objects, each object is executed as a separate thread to improve execution time. The agent also monitors the current load of the host. If it exceeds a certain threshold value, the agent invokes the migration daemon to migrate some or all objects to another idle or lightly utilized host in the system. Figure 4 shows the internal architecture of the agents. The communication daemon uses the communication port to interact with the root server and other agents in the system. The RMI daemon dynamically loads the object stubs and binds them to a RMI registry. It also intercepts any RMI call when a specific object being migrated. As described earlier, the migration daemon uses the migration port for transportation of serialized objects between agents. The remote I/O daemon uses the communication port for input purposes and the remote output port for output purposes. In AdJava, three different kinds of communication between the root server and the agents are possible. In the current version of AdJava there is only one root server, so there can be no communication between root servers. At start-up, the root server performs handshaking with agents to check that the agents are alive. After successful handshaking the root server passes object information to the agents so that agents can create corresponding objects in remote hosts. Once an object finish running, the root server may ask the agents for timing data and any result from the distributed object. On handshaking, the agent sends the initial load information to the root server. Once an object is migrated to another host, agents intercept all incoming calls to the migrated objects through a proxy object and raise an exception which details the new location of the object. Agent-agent communication occurs during migration and also if there is any inter-object communication, which should be as minimum as possible for better execution time.

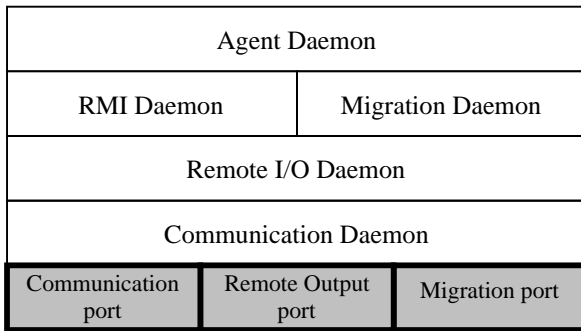


Figure 4: Internal architecture of the agents.

3.8 Migration

An important feature that is not readily available in many systems designed for implementing distributed applications is the ability to easily migrate computation, especially within heterogeneous environments. To migrate an object, some state information has to be saved and shipped to the destination host. At the target destination, the state of the object must be reestablished and execution of the object needs to be rescheduled. The state of a Java thread, which is internal to the JVM, consists of several items (Meyer and Downing 1997) discussed below:

- *Java Stack*: It stores *frames*. A new frame is created each time a Java method is invoked. A frame is destroyed when a method completes.
- *Code*: The set of Java classes which includes a Java method currently being executed by the thread.
- *Data*: The contents of the instance variables of the objects.

In order to migrate a thread, its execution must be suspended and its Java stack must be captured and transformed into a serializable format and then sent to the target location. At the target location, the stack must be reestablished to start execution from the same program counter location. If migration exhibits this property, it is called *transparent* or *strong* migration. If the programmer has to provide explicit code to read and reestablish the state of the object, it is called *non-transparent* or *weak* migration (Fünfroeken 1998). Although code migration and data migration is strongly supported by Java, thread migration is completely unsupported by the current JVM. Java threads are not implemented as serializable. Furthermore, the Java language does not define any abstractions for capturing and reestablishing the thread information inside the JVM. With Java, it is difficult to implement a transparent migration mechanism without degrading portability or efficiency for several reasons (Te 2000):

- If the JVM is extended to support migration (Bouchenak 1999 and Peine et al 1997), then the performance of the migrating program would be nearly the same as before. However, the migrating program loses its portability because it can only be executed on those modified JVMs.

- If the migrating mechanism is provided as a class library (Izzat et al 1999), it will provide nearly the same benefits as using a modified JVM. Unfortunately it has one major obstacle that is impossible to overcome in the current version of Java. Transparent migration requires transmission of the stack. Java forbids dynamic inspection of the Java stack and therefore it is not possible to transmit the stack.
- The alternate approach (Fünfroeken 1998 and Sekiguchi et al 1999) is to transform the Java source program in such a way that, it can explicitly manage its execution state as a Java Objects, which enables transparent migration. The major drawback is a slowdown in execution speed due to a code fragment being inserted to maintain migration information.

AdJava uses source level translation that preserves Java's portability. The transformed code uses only Java and Java/RMI and it does not require any changes to the JVM to migrate objects between hosts.

3.8.1 Object migration in AdJava

To support transparent migration, the *run* method in the thread must be suspended and the object holding that thread must be transmitted to the new location and then be resumed. The following sections describe the migration scheme used in AdJava that is based on source level transformation. This section gives a brief overview of AdJava's migration process.

To capture the state of a Java program, it is necessary to access the method call stack along with the program counter. However, Java does not support capturing all of the call stack or the program counter. There are a few systems (Bouchenak 1999) that can capture the stack and the program counter. However these systems have modified the JVM to provide this support. In contrast, AdJava uses a method that does not require the modification of the JVM. To capture the state of a Java program, the pre-processor instruments the user code by adding additional codes that deals with the capture of the actual state and its re-establishment. However, inserting code to automatically extract state information and restart at arbitrary points during execution is extremely difficult and requires a major extension to the compiler or the interpreter; example includes Ara (Peine and Stolpmann 1997) and JavaGo (Sekiguchi et al 1999).

AdJava restricts the points in the execution from which execution can resume. This not only greatly simplifies the state capture mechanism but also makes the implementation very efficient. As user objects in AdJava are threaded, the pre-processor only needs to instrument the *run* method of the object for migration purpose. Therefore the state is saved before any method is invoked inside the *run* method. So, if the object is suspended inside another method, then in the destination host, execution resumes from the beginning of that method.

3.8.2 Saving the execution state

Java's serialization mechanism provides a way to save the state of an object. This state consists of the value of all class and instance variables inside that object. Therefore, using serialization, most of the language level information required to reestablish the program state can be captured. However, the information regarding method call stack and the program counter is missing. As AdJava only resumes inside the *run* method and not within other methods, only the program counter information is necessary to restart the object. Therefore the pre-processor introduces a new variable *level* and inserts it into the *run* method to simulate the program counter. The newly introduced program counter is incremented after each statement, so that when restarted, execution can resume at the point where it was suspended. The challenge of inserting an artificial program counter lies in the compound statements. Therefore the pre-processor rewrites the compound statements in a form which allows AdJava to suspend and resume an object at any point within a compound statement.

3.8.3 Transmitting the state

Once the object is suspended and its state is being captured, it is serialized and saved to a file. Then the saved file along with the *codebase* is transmitted to the destination agent through a TCP socket connection. After successfully transmitting the suspended object, the source agent creates a corresponding proxy object and destroys the original object just being migrated.

3.8.4 Resuming

To enable a migrated thread to resume execution, a mechanism that enables execution to jump to any position inside the run method is needed. However, Java does not provide a jump mechanism. The *break* and *continue* statements in Java permit only to escape from a block. The scheme used in AdJava to jump between statements is implemented by using *switch..case* statements. This scheme is based on transforming the *run* method into a form in which the method can be resumed from any program point. Along with transforming the *run* method, the preprocessor also inserts a new method named *reStart*. So, when an object is migrated to the destination machine the agent there invokes the *reStart* method. It first changes the value of the *moved* variable to *true* and starts the thread using Java API's *start* method.

3.8.5 Reference updating

As objects are migrated between different host, there must be a way to ensure that, references hold by an agent, still works after migration has been completed. AdJava uses a transparent mechanism to update the reference of an object once it been migrated. In AdJava, Each agent keeps a table or the *codebase*, to store the address of the agents and information about objects that are bind to those agents. When an object is migrated the source agent does not delete that entry from its codebase, rather it creates a proxy object and binds it with the same address as the migrated object. Therefore, the proxy object

behaves like the original object. However, the difference is, when any call is received by that proxy object, it throws an *MigratedException* back to the caller. Before throwing the exception, the agent embeds the new address of that object into the exception. So, on the other side, if the caller gets an *MigratedException*, it updates its reference with the address inside the exception.

3.8.6 Limitations

Currently, there are some limitations in the AdJava's implementation of its migration scheme. Most of the limitations are essentially due to source-level transformations. Therefore, similar limitations apply to other migration schemes (Fünfroeken 1998 and Sekiguchi et al 1999) based on source-level translation. Instrumentation and insertion of code introduces time and space overheads. See Section 4.4 for cost relating to migration.

As it is already been mentioned, AdJava can resume execution only inside the *run* method and not within other methods. So, if an object is suspended inside some other method, the system loses the computation done within that method, because when it is resumed, the same computation is again performed. Currently, all *while* and *do..while* loops are transformed to *for* loop. However, if the condition in the loops is not numerical, then those loops are not transformed and therefore, in those cases, the thread only resumes at top of the loops.

4 System evaluation

In order to demonstrate the practical relevance of AdJava, the system is evaluated with two different applications. These results show that AdJava provides good speedup in each case.

4.1 Testbed

All experiments were conducted between various combinations of several SPARC and Intel machines. Most of the SPARC machines have either dual or quad processors. The Intel machines have one processor in them. Processor speed ranges from 250 MHz to 450 MHz. All of these machines have enough disk space. All of the machines are running on Solaris 8. Machines are connected by a mixture of ATM, Fast Ethernet and Ethernet networks. All experiments were conducted using SUN's JDK version 1.3.0.

4.2 The Dinning Philosophers problem

The classic Dinning Philosophers problem was implemented and run on a single machine. This application was chosen because of its concurrent and distributed nature. The implementation of the dinning philosophers problem was such that all philosophers eat five times before terminating. When each philosopher is eating and thinking, they are actually multiplying two, 200 x 200 integer matrices. The algorithm used to implement the dinning philosopher problem avoids any deadlock and starvation problems. Specifically, all Fork objects are implemented as monitors and a Philosopher can only pickup two forks at a time if other Philosophers

are not using them. All the philosopher objects were running on different threads. The machines chosen to run the Dining Philosophers problem, each have one processor in them. The special *Distribute* keyword is used to produce a distributed implementation. The resulting program is run using the AdJava system. Figure 5 shows the results of these tests. When the program is running on a single machine, it runs as a serial program as the machines have only one processor in them. When the philosopher objects are distributed over different number of machines, the execution time decreases correspondingly. Therefore the system shows a near linear speedup from the Dining Philosopher problem. The setup time, which consists of the pre-processing time, handshaking time, stub/skeleton uploading time and RMI call time, increases gradually by a fraction of milliseconds. This is due to the fact that, as number of machine increases, the time to handshake and the time to upload stubs to the remote agents, increases accordingly.

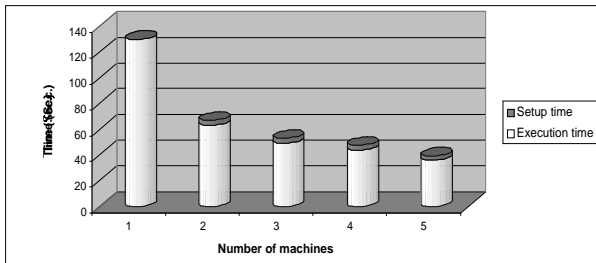


Figure 5: Execution time of Dining Philosophers problem.

4.3 The Matrix multiplication problem

The Matrix Multiplication problem was chosen because many scientific applications either use matrix multiplication directly or must solve similar numeric problems that are highly regular in their structure and require significant movement of data between computations. Hence, matrix multiplication represents a good test case for studying AdJava capabilities in this area.

The simplest possible implementation of matrix multiplication on a single machine is a triply nested loop, where the outer (i) and middle (j) loops iterate over the rows and columns of the two matrices and the inner-most loop (k) computes the inner product of a row and column as follows: $C[i,j]=C[i,j]+A[i,k]*B[k,j]$. To distribute the computations over a number of objects, the two $m \times m$ input matrices A and B and the resulting matrix C are all partitioned into rectangular blocks of size $s \times m$, where $s=m/\text{number of objects}$. In all the versions of the matrix multiplication problem there are ten objects. In the parallel and the distributed version, there are ten threads multiplying each block concurrently. Therefore using the *Distribute* keyword, corresponding objects can be distributed over number of machines, which can compute its portion concurrently.

Figure 6 shows the result of these tests. A quad processor machine is chosen to run the serial version of the matrix

multiplication problem. The same machine is used to run the parallel version of the matrix multiplication problem. It is evident from Figure 6 that, parallel version of the matrix multiplication problem showing better execution time then the serial version. Machines chosen to distribute the same parallel version of the matrix multiplication program had either one or more than one processor in them. As the program is distributed over more machines, the corresponding execution time improves accordingly. As number of machine increases,

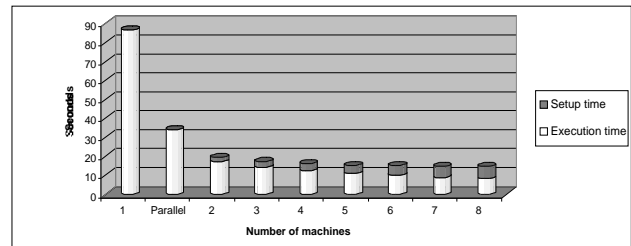


Figure 6: Execution time from the matrix multiplication problem.

the corresponding setup time is also increases. This is due to the fact that, as number of machine increases, the corresponding handshaking time and stub/skeleton download time increases that in turn increase the setup time.

4.4 Cost of migration

One of the major features of AdJava is the ability to automatically migrate objects from a highly loaded machine to a lightly loaded machine. Figure 7 shows the time to migrate objects of different sizes. In conducting this experiment, an object which comprised of a two dimensional array was migrated from one machine to

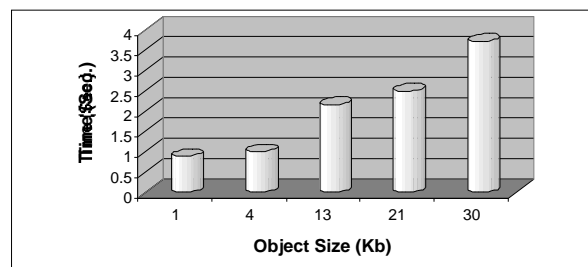


Figure 7: Time to migrate object.

another by artificially increasing the load of one machine. All times were taken from the point when the load exceeds a certain threshold and to the point when that object is resumed. It is clear from Figure 7 that, the cost of migration depends on the size of the object. This is because, migrating an object requires that the object to be serialized, transferred to the target host and then deserialized. So most of the time is spending serializing and deserializing the object and as discussed by Phillippsen et al (2000), Java's serialization process is too slow. So with a better serialization and deserialization process, AdJava will provide better migration time. Additionally, when the object is migrated to a new host, class loading is required which also takes time.

4.5 Size of transformed code

The current design and implementation and resulting performance has been greatly influenced by AdJava's goals to produce a automatic and transparent system that is easy to use and that does not change the Java Virtual Machine. In doing that, AdJava has to insert additional

Problem	Code size (KB)		
	Original	Modified	growth ratio
Dinning Philosophers	3.53	14.00	3.96
Matrix multiplication	3.36	12.88	3.83

Table 1: Code growth in AdJava.

codes into the user program. Instrumenting and inserting code introduces time and space overheads. Since the pre-processor adds code to the program, there is always a file size space penalty. At run time this introduces a memory space penalty. Also to accommodate RMI specification, several new interfaces have to be created and corresponding stub and skeletons are generated. That increases the number of files in the system. All these factors are responsible for AdJava's code inflation. Table 1 shows the corresponding code growth for the two implemented applications. We believe that, the performance and the amount of features along with the ease of use of AdJava, this growth ratio is reasonable.

4.6 Discussion

Having discussed the design and implementation of AdJava and presented a range of performance results, this section discusses issues related to AdJava's performance.

Although the mechanism and functionality provided by AdJava is sufficient to support different variety of applications, the system will continue to evolve. The current design and implementation of AdJava has been greatly influenced by the goal to produce a system, that is transparent and easy to use and that does not change the Java Virtual Machine. These decisions and choice of Java have influenced the performance of AdJava.

Some of the performance overheads are due to the way Java executes programs, i.e. interpretation rather than compilation. In the setup phase, nearly all the time is being spent on compiling the resultant class files and generating the stubs and the skeletons for the remote objects. Dynamic class loading also consumes much of the time. It is possible to improve the system by sending only one compressed JAR (Sun Microsystems 2001) file that contains all the stubs and skeletons. The important result here is that, the total execution time suggests that AdJava is providing better performance than a concurrent implementation running on a single machine. However to achieve this speedup, the programmer does not need to do anything other than employing one new keyword in the program.

5 Conclusion

This paper has presented a system called AdJava that implements a novel model for distributed computing using Java. It requires minimal interaction between the system and the programmer to achieve improved performance. AdJava makes four main contributions to the area of Java-based distributed object programming.

- AdJava is 100% Java compatible. No changes to the Java Virtual Machine are required.
- AdJava allows any Java object to be used in a distributed fashion by adding the *Distribute* keyword in front of it.
- AdJava transparently migrates running objects to other lightly loaded machine to load balance the system, albeit in a limited fashion.
- AdJava includes a significant number of features, which, to the author's knowledge, have not been previously combined into a single Java-based distributed environment.

AdJava uses the computing power of the lightly loaded computers on the network by providing a transparent approach through the use of software agents. AdJava alleviates the complexities and problems associated with manual distribution and provide better throughput and execution time by managing all of the distribution issues through the use of autonomous software agents. AdJava allow users to seamlessly create objects on different machines and users can access these objects in a fashion similar to local objects. In AdJava, distributed objects can produce output to files and consoles in a similar manner to that of a local object. In AdJava, in situations of heavy load, the agents transparently migrate remote objects as necessary. The system provides all this support without any programmer involvement and without modifying the Java Virtual Machine.

6 Reference

- ALEXANDROV, A., IBEL, M., SCHAUSER, K.E. and SCHEIMAN, C. (1997): SuperWeb: Research issues in Java-based global computing, *Concurrency: Practice & Experience*, 9(6): 535-553.
- BARATLOO, A., KARAU, M., KEDEM, Z., and WYCKOFF, P. (1996): Charlotte: Metacomputing on the Web, *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*.
- BIRREL, A. D., and NELSON, B. J. (1984): Implementing remote procedure calls, *ACM Transactions on Computer Systems*, 2: 33-59.
- BOUCHENAK, S., (1999): Pickling threads state in the Java system, *Third European Research Seminar on Advances in Distributed Systems*, Portugal.
- CHRISTIANSEN, B., CAPPELLO, P., IONESCU, M.F., NEARY, M.O., SCHAUSER, K. and WU, D. (1997): Javelin: Internet based parallel computing in Java, *ACM 1997 Workshop on Java for Science and Engineering Computation*.

- DETMOLD, H., HOLLFELDER, M. and OUDSHOORN, M.J. (1999): *Ambassadors: Structured Object Mobility in Worldwide Distributed Systems*, 19th IEEE International Conference on Distributed Computing Systems, Austin, Texas, 31 May–5 June, 1999, pp 442–449.
- FALKNER, K.E.K., CODDINGTON, P. and OUDSHORN, M.J. (1999). *Implementing Asynchronous Remote Method Invocation in Java*, 6th Australian Conference on Parallel and Real-Time Systems, Melbourne, 29 November–1 December 1999, Springer-Verlag, pp. 22–34.
- FÜNFROCKEN, S. (1998): Transparent migration of Java based mobile agents (Capturing and re-establishing the state of Java programs), *Proceedings of the Second International Workshop on Mobile Agents*.
- IZZAT, M., RECHT, T. and CHAN, P. (1999): Ajents: Towards an environment for parallel, distributed and mobile Java applications, *ACM 1999 Java Grande Conference*.
- LANGE, D.B. and OSHIMA, M. (1998): Programming and deploying Java mobile agents with Aglets, Addison Wesley.
- MAASSEN, J., NIEUWPOORT, R. V., VELDEMA and BAL, H.E. (1999): An efficient implementation of Java's remote method invocation, *Proceedings of ACM Symposium on Principles and Practice of Parallel programming*.
- MESSAGE PASSING INTERFACE FORUM, (1994): MPI: a message passing interface standard, <http://www-unix-mcs.anl.gov/mpi/>.
- MEYER, J. and DOWNING, T. (1997): *Java Virtual Machine*. Cambridge, O'Reilly.
- OUDSHOORN, M.J. and DETMOLD, H. (2000): *Ambassadors: A Communication Structure for Mobile Java Objects*, Scuola Superiore G. Reiss Romoli, SSGRR-2000, in L'Aquila, Italy, 31 July – 6 August, 2000. CD-ROM, 12 pages, invited paper.
- PEINE, H., and STOLPMANN, T. (1997): The architecture of the ara platform for mobile agents, *Proceedings of the Second International Workshop on Mobile Agents*.
- PHILLIPSEN, M. and ZENGER, M. (1997): JavaParty: Transparent remote objects in Java, *ACM 1997 Workshop on Java for Science and Engineering Computation*.
- PHILLIPSEN, M., HAUMACHER, B. and NESTER, C. (2000): More efficient serialization and RMI for Java, *Concurrency: Practice and Experience*, 9(11): 1225-1242.
- RAJE, R., WILLIAM, J.I. and BOYLES, M. (1997): An asynchronous remote method invocation mechanism for Java, *ACM 1997 Workshop on Java for Science and Engineering Computation*.
- SEKIGUCHI, T., MASUHARA H. and YONEZAWA A. (1999): A simple extension of Java language for controlable transparent migration and its portable implementation. *Lecture Notes in Computer Science*.
- STRABER, M., BAUMANN, J. and HOHL, F. (1996): A Java based mobile agent system. *ECOOP '96 Workshop on Mobile Object System*.
- SUN MICROSYSTEMS (2001): *Remote Method Invocation specification*, <http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html>.
- SUN MICROSYSTEMS (2001): *JDK 1.3.02*, www.java.sun.com.
- TANDIARY, J.F., KOTHARI, S. C., DIXIT, A. and ANDERSON, E. W. (1996): Batrun: Utilizing idle workstations for large-scale computing, *IEEE Parallel and Distributed Technology*, 4(2): 41-48.
- TE, D. S. (2000): Approaches to Capturing Java Thread State, <http://citeseer.nj.nec.com/262653.html>.
- THIRUVATHUKAL, G.K., THOMAS, L.S. and KORCZYNSKI, A.T. (1998): Reflective remote method invocation, *Concurrency: Practice and Experience*, 10(11-13): 911-925.
- XU, C. and WIMS, B. (2000): A mobile agent based push methodology for global parallel computing, *Concurrency: Practice and Experience*, 12: 705-726.
- YU, W. and COX, A. (1997): Java/DSM: A Platform for Heterogeneous Computing, *Concurrency: Practice and Experience*, 9(11): 1213-1224.