# A Distributed Object-Oriented Parallel Computing Environment Based on Java/CORBA

Chi-Chang Chen*
Information Engineering Department
I-Shou University
Kaohsiung County, Taiwan, ROC

Kin-Hsing Hsieh
Computer Science & Engineering Department
Tatung Institute of Technology
Taipei, Taiwan, ROC

**Abstract** *In this paper, a distributed object oriented parallel computing environment, which is called Pax system, based on Java/CORBA is proposed. The Pax system sets up a virtual parallel-computing platform, which is built on heterogeneous hardware and software systems that run Java Runtime Environment and CORBA package. Standard Java/CORBA developing process is used to develop programs on the Pax system so that it can reduce the learning time of parallel computing programming.*

*Keywords:* distributed parallel computing environment, distributed system, parallel computing, CORBA, Java programming.

## 1 Introduction

Object-oriented design and programming is one of the major trend of software development in this decade. In the object-oriented paradigm everything, including methods and data, is encapsulated in objects, and a program is merely the interaction between objects.

It is a big challenge to coordinate the tasks that are performed in different hardware and software platforms. Currently, there are two major organizations working on the heterogeneous object-oriented distributed computing in effort of resolving the incompatible protocols using in the network. One of the results is called Object Management Architecture (OMA) by the non-profit Object Management Group (OMG) [10, 9]; and the other one is called Distributed Network Architecture (DNA) by the giant software vendor, Microsoft. Both of the specifications deal with the problem of coordinating tremendous objects residing in different platforms.

In this paper, we concentrate on the software emulation of parallel machines. The whole mechanism is built on the joint effects of Java and Common Object Request Broker Architecture (CORBA), which is the communication specification defined by OMG. We develop a distributed object-oriented parallel computing environment called *Pax system*. The Pax system consists of Pax machines and Pax class library. Both of them apply the concept of *object group design pattern* [15, 16], which provides object management functions and a synchronization mechanism called *virtual synchrony*. Each Pax machine is an independent process runs on the host system. The center of a Pax machine is the administrator object and it's associated graphic user interface. The main function of Pax class library is to map the static interconnection network configurations to the virtual software emulation in the Pax system. Hence, the traditional parallel computing programs that is designed to run on the interconnection networks can be adopted to the Pax programs easily.

The remaining of this paper is organized as follows. Section 2 describes the architecture and the design of the Pax system. Section 3 discusses the implementation issues and the

*Supported in part by the National Science Council under the grant NSC 88-2213-E214-040

tradeoff. Applications and performance evaluation are presented in section 4. Section 5 concludes the paper.

# 2 Architecture of the Pax System

The Pax system can be divided into three parts in concept. They are Pax machines, Pax class library and the computational concept behind the scene.

## 2.1 Pax Machines

The Pax system consists of individual Pax machines that are resident on separate Java Virtual Machines(JVMs) in a distributed heterogeneous network environment. Each machine represents an independent computing space and connects to each other through the Java ORB. The major task of a Pax machine is to run a concurrent Pax program. In the modern operating system, one physical computer can support multiple processes; consequently, multiple Pax machines could be mounted on a real computer.

Each Pax machine consists of fundamental Java classes including PaxObjectDSI and AdministratorDSI, which are dual skeleton implementation classes, and PaxAdm. PaxObjectDSI, which implements the group design pattern, is the base class of most of the classes in the Pax machine. AdministratorDSI takes care of several functions, such as network class loading, class files spreading, Pax machine performance evaluation, and Pax program launching. PaxAdm implements the Graphic User Interface(GUI) Monitor window reporting the status of all the Pax machines in the Pax system for the Administrator. The dual skeleton implementation (DSI) issues will be discussed in next section.

### 2.1.1 PaxObject Interface

The PaxObject interface is the most important interface of the Pax system. It represents the modified object group design pattern called the *group-ware*. The *object group design pattern* was proposed by Silvano Maffeis in [16] for the fault-tolerance and the group communication in a distributed system. Object group design pattern provides the concept to handle a group of objects distributed across networked machines. If an object crashes, other object in the same group can take charge immediately. The multicast and broadcast communication between objects within the group is based on the virtual synchrony model in which the states of objects that are belonged to the same group are consistent. In order to maintain the consistent view of a group, the new joining object would receive the common state and the changes to the state would broadcast to every member in the group. Therefore, if an object is requested to provide some information of the state of the group, it can perform this request by itself without notifying other objects.

Although in the Pax system the objects of the same group do not have to keep the same state in the design of *group-ware*, they are still capable of doing other operations in the object group such as join the group, depart the group, and perform virtual synchrony broadcast. In addition to the benefit of fault tolerance, group-ware design also facilitates the distribution of concurrent subtasks.

There is a unique master object, called the *coordinator* of the group, which is responsible for the communication within the group. The coordinator has the total ordering list of members. If any member wants to broadcast or multicasts information in the group, the message needs to send to the coordinator first; then, the coordinator broadcast or multicast the message by the sequence according to the total ordering list.

To maintain the total ordering list of members is the primary task of the coordinator. Once the event of joining or leaving occurs, the coordinator has to update the content of the list.

If there are a lot of groups needed to deal with, how could an object finds out what group had already existing in the Pax system and how to find its coordinator? The solution is by using a hash table that maps the group name to the associated coordinator object. An object called *Omni* takes charge of it. Pax programs can register a new relationship of the coordinator and its group name to *Omni* or cancel the relationship dynamically.

#### 2.1.2 Administrator Interface

The Administrator interface, which is derived from PaxObject interface, has several functions, such as network class loading, class files spreading, performance evaluation, and Pax programs launching. The Java duel skeleton class AdministratorDSI, which implements the Administrator interface defined in IDL, is also derived from PaxObjectDSI class. Why should the Administrator inherit from PaxObject? The answer is straightforward, because all the administrators of Pax machines, which own its identical name in the Pax system, also group themselves together in order to keep tracks of other members that form the Pax system.

In the Pax system, the users just need to place their own Java class files in one Pax machine. A Pax machine may share the same file system with other ones. When a Pax program launches on a Pax machine, it may distribute subtasks to other Pax machines through the administrator. Therefore, other Pax machines requested by the administrator to run the Pax program may need the same Java class files to execute. If the default JVM class loader cannot find the bytecodes it needs, the PaxClassLoader will be called to get them from the Pax machine that wants to spread them.

#### 2.1.3 PaxAdm Interface

Finally, the administrator object has to provide connections to its GUI representation objects. Thus, a simple notification mechanism is designed for the administrator interface and the associated administrator monitor interface. This PaxAdm Java class represents the GUI tool. The PaxAdm class implements the Java interface AdministratorMonitor, which maps to the corresponding IDL interface. In our current implementation, one PaxAdm object is corresponding to one Pax machine object. The PaxAdm class is designed as an integrated visual interactive tool for management. User can start a PaxAdm object to create a Pax machine.

### 2.2 Pax Class Library

In order to run the programs designing for some specific interconnection networks, such as hypercubes and trees, the Pax class library provides two fundamental classes for mapping the nodes from an interconnection network to the Pax machines. These two fundamental classes are Node and NodeManager. The flexibility of deploying nodes into a single Pax machine is achieved by the cooperation of these two classes or, to be more specifically, their subclasses.

The Node interface, which inherited from the PaxObject interface, represents the metanodes of the static interconnection networks. Normally, each node of a static interconnection network has its identical label string that can express the relationship or connectivity to other nodes.

The NodeManager interface is also inherited from the PaxObject interface. The NodeManagerDSI is the implementation of Java class corresponding to the IDL NodeManager interface. For each Pax program, one NodeManagerDSI object will be deployed to manage all Node objects that are in the same computing space.

### 2.3 The Underlying Concepts of Group-Ware

In the design of PaxObject interface, which is the root interface of most other Pax system interfaces, every PaxObject instance can become the member of a group. Objects of the same group have the same functionality, and therefore they can share the distributed jobs, or support each others in case of object or machine crash. There is a coordinator in each group. The coordinator object is responsible for maintaining a list of members in the group and dealing with the arrival and departure of members.

When an object wants to join a group, it should check whether the group exists. If the group had registered to *Omni*(the class that manages all the groups), the object should ask *Omni* to return a reference of the coordinator object corresponding to the group, then invoke the join handler of the coordinator object. Otherwise, the object becomes the first member of the new group, and therefore it is the coordinator as well. After handling the join operation of itself, it should register the associated relationship between itself and the group name to Omni.

In addition to the join and leave operations, the object group-ware supports virtual synchronized broadcast. Any two members among the same group have an order that is according to their Java hash-code. When the coordinator calls the numeration method of the hash-table class, a fixed ordered list will be generated.

# 3 Implementation Issues

## 3.1 Dual Skeleton Implementation

According to Java syntax, a class only can have one direct super class although one could implement as many interfaces as he/she wishes. This causes a major constraint of code reuse in Java/CORBA mapping.

CORBA is a client/server style mechanism. Requests, which are issued from the client side, are passed from the stubs to the skeletons and the results return in the opposite direction. The processes are available for both static and dynamic method invocations from stubs to skeletons, but only for the static method invocation from a skeleton to another skeleton. Therefore one skeleton must use static method invocation mechanism to request other skeletons services. This means that the type of server stubs must be determined at compile-time and thus suffers the loss of flexibility.

Due to the problems mentioned above, the consequence of building a large scale class library is to combine static and dynamic implementation into a mass. Each Java class has to subclass from Dynamic Implementation class and implements its corresponding <Interface>Operations interface which is generated by the idl-to-Java compiler. Therefore, a subclass could inherit from a class and implement its own corresponding operations interface at the same time.

The inheritance relationship in IDL is just the interface inheritance, not the implementation inheritance. However, in reality, what programmers need are both inheritance relationships. The other question is that the derived interface can redefine its properties that are inherited from its ancestor interfaces in the OMG IDL. In the dual skeleton implementation, programmers who code the implementation class corresponding to the derived interface own the choices of calling the methods of its ancestor implementation class. According to the nature of object oriented paradigm, the derived version of methods will be called before its base version when the method invocations happened. If the redefinition occurs, the redefined methods can modify or drop the calls to direct base dual skeleton class; therefore, the redefinition would be achieved in the implementation side, too. This trick is quite important because the derived class should own the initiative to decide doing something in a brand new way or following traditions or just mixing both approaches.

## 3.2 Synchronization

Java supports multithread programming. For Java, there are two approaches to deal with the synchronization between threads. Both use the keyword, *synchronized*, to set up a critical section. If the keyword *synchronized* applies to a method of an object, the whole section of the method would become a critical section; otherwise, programmers can use the keyword synchronized to maintain a critical section in any method. The discussion above is merely meaningful in one JVM. For more than one

JVMs, two distributed synchronization mechanisms could be applied to the system. One is the *virtual synchrony model*, and the other is the *barrier operation*.

As mentioned before, the object group pattern use the coordinator to broadcast or multicast messages among the objects in a totally ordered grouped set. Under this circumstance, each grouped object should own a method that is triggered by the coordinator to receive messages. Programmers should apply the spirit of event driven model to their main independent threaded objects. The receive methods are used to setup some flags inside the object, and the main control stream of the system should be an infinite loop to response according to the flags, which decide the flow of control. Inevitably, the hot waiting due to the infinite loop is the source of inefficiency.

The barrier operation is also a synchronization operation used very often in a distributed system. The barrier operation sets up a barrier fence in front of a set of independent threads. When the execution of thread meets the barrier fence, the thread will not be continued until all of blocked threads of the same barrier operation arrive.

In the implementation, there exists a coordinator of a barrier group which is composed by threaded objects that want to perform the barrier operation. Every object in this barrier group should call the barrier operation method of the coordinator. The thread that is attached by the barrier operation coordinator object should be independent to other threads that are attached by other objects that performed the barrier operation. The barrier operation coordinator object keeps a list of the members of the barrier operation. What the barrier operation method does is an infinite loop which checks the flags representing the callings of members. If every object among the barrier group calls the barrier operation method of the coordinator, the coordinator would exit the busy waiting loop inside the method; therefore, the execution flow of every object among the barrier operation resumes.

## 3.3 Distributed Garbage Collection

In the JVM, there is a separate and low priority background thread called Java Garbage Collector that takes the responsibility for deallocation memory. The Java runtime system keeps an eye on the memory allocation events and monitors the references for individual objects.

In the simple environment inside the JVM, the garbage collection is quite simple, but it becomes complicated in a distributed system. The program objects may be used in a single program on a single JVM or be shared by many programs that are distributed on many Java Virtual Machines resident on the Internet. Due to the nature of network, slow connection and platform crashes of both clients and server sides cause additional problems to the garbage collector. For example, if the client platform crashes suddenly, the server objects that are still alive and waiting for responses from the client program should be deactivated and freed.

CORBA uses both reference counting and Internet connection management mechanism [6]. From the moment of instantiation of a server object, the reference count number of it is implicitly increased whenever a new reference pointing to it, and decreased whenever an existing reference is destroyed. If the reference count equals to zero, the instance of the server object would be cleaned up by the garbage collector. This mechanism is not only practiced on the Java ORBs but also implemented on other CORBA-compliant ORBs no matter what programming language they are supported and whether do they have garbage collection mechanism. In addition to the implicit ways to count the reference, CORBA provides explicit ways for adding and removing references, called *duplicate* and *release*. These two operations are tied into the definition of the root interface, org.omg.CORBA.Object, which is the origin of all CORBA interfaces.

```
interface Object {
    ::CORBA::Object _duplicate();
```

```
void _release();
...
}; // Partial IDL definition of org.omg.CORBA.Object
```

When a new reference is copied, the duplicate operation should be called to indicate that a new reference is created; on the other hand, the release operation should be called in order to perform some clean up procedures when the programmers want to discard an existing reference.

The implicit and explicit ways discussed above merely guarantee that the garbage collection mechanism runs correctly under normal network connection circumstances; but it does not work in the case that are mentioned in the previous example. Consequently, CORBA takes care of the network connection to handle the connection failure problems and slow transmission introduced lag issues. When the ORB and the BOA observe these two kinds of phenomena, any references that held by the client program would be called to execute release operations. As a result, if the connection between the client side and server side keeps in touch, the references will not be released; thus, the server objects would not be collected by the garbage collector.

Furthermore, there is a more straightforward way to vanish the server objects that are useless. Programmers can take advantage of the deactivate_obj operation of the BOA interface to deactivate it.

# 4 Experiments

## 4.1 Programming and Execution of a Pax program

A Pax program is a program designed for the distributed object oriented parallel programming on the Pax machine. In addition to using the Pax class library package, the programmer can use the Java Core Class Library, and the CORBA packages as well.

The programming can adopt from the traditional parallel programming for the static interconnection networks. Since there are already a lot of parallel programs design for them, in order to reuse those parallel programs that are usually dedicated to specified interconnection network, the programmer can subclass the specified Node classes in the Pax class library and embed the program in it.

To launch a Pax program, we first run the Monitor GUI program. In the Monitor window, we click the **launch** bottom and fill out the name of the Pax program. Then, the local administrator will take charge of the task allocation according to the request of the Pax program.

## 4.2 Experiments

In the experiments, we implement a parallel Quicksort algorithm[?] on the Pax system. We compare the performance among several configurations. We use Visigenic VisiBroker version 3.2 evaluation version for x86 instruction set [11, 1] and Sun's Java development kit version 1.1.5 [5, 1] on these three Pentium PCs. Two of the three machines are installed Windows NT version 4.0; the other is installed Windows 95 operating system. One Pax machine is forked in each real computing machine and one SortNode object is running so each Pax machine.

We compare the execution time for sequential Quicksort, parallel Quicksort on two Pax machines, and parallel Quicksort on three Pax machines of data sizes vary from 100,000 elements to 800,000 elements. The experimental result is shown in Table 1.

As expected the execution time of Quicksort on three Pax machines is less than the other two cases. However, due to the communication and synchronization overhead for the distributed system, the superiority is not very outstanding.

Undoubtedly, the communication overhead of the Pax system is larger than that of the comparable size parallel machine, and communication usually implies synchronization. One of our future work is to improve the performance of the Pax system. For the program

Table 1: Experiment results of Quicksort on various configurations. (in microsecond)

| Number of Data | Sequential (without CORBA) | Two Pax Machines | Three Pax Machines |
|---|---|---|---|
| 100000 | 2040 | 2374 | 2283 |
| 200000 | 4450 | 4577 | 3945 |
| 300000 | 6820 | 6640 | 5939 |
| 400000 | 9390 | 8943 | 7891 |
| 500000 | 11870 | 11190 | 10355 |
| 600000 | 14330 | 13710 | 11617 |
| 700000 | 17020 | 14720 | 13920 |
| 800000 | 19660 | 17600 | 16132 |

design, in order to make distributed Pax programs more efficient, the part of communication should be reduced as much as possible, and therefore it will also reduce the overhead due to synchronization.

## 5 Conclusion

The Pax system makes the traditional parallel programs easy to be reused on the distributed systems. The Pax system can also reduce the training and learning cost of programmers that have no parallel programming experience but familiar with object oriented programming. Although the performance of the Pax system may be not as good as other distributed parallel computing environments such as PVM, the advantages are its simplicity and availability. Since the Pax system is based on the standard Java and CORBA software packages, they are easy and cheap to acquire from the internet. Hence, the Pax system can be deployed smoothly without complicated configuration problems to almost every hardware and software platform available. Therefore, the use rate of computers grouped by LAN can be raised. Besides, we believe that the knowledge to program Pax programs is also useful to develop other distributed client/server applications that are based on CORBA.

## References


[1] Jamie Jaworski, Java 1.1 Developer's Guide, Sams.net Publishing 1997.

[2] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, Introduction to Parallel Computing : Design and Analysis Of Algorithms, the Benjamin/Cummings Publishing Company, 1994

[3] Silvano Maffeis, PIRANHA : A Hunter of Crashed CORBA Objects, Department of Computer Science Cornell University, February 1996.

[4] Silvano Maffeis, The Object Group Design Pattern : an Object Behavioral Pattern for Fault-Tolerance and Group Communication in Distributed System, Department of Computer Science Cornell University, February 1996.

[5] Jeff Nelson, Distributed Garbage Collection : Using CORBA, RMI, and DCOM to solve an increasingly tough problem, Java Developer's Journal, Volume 2 Issue 9 Page 46.

[6] Robert Orfali, Dan Harkey, Client/Server Programming with Java and CORBA, John Wiley & Sons, 1997.

[7] Object Management Group, CORBAservices: Common Object Services Specifications, 1997.

[8] Object Management Group, The Common Object Request Broker: Architecture and Specification Revision 2.2, February, 1998.

[9] Andreas Vogel, Keith Duddy, Java Programming with CORBA, John Wiley & Sons, 1997.