University of Alberta

Library Release Form

Name of Author: Alberto Daniel Zubiri

Title of Thesis: An Assessment of Java/RMI for Object Oriented Parallelism

Degree: Master of Science

Year this Degree Granted: 1997

. . . .

Alberto Daniel Zubiri
11144 28 Avenue Apt.303
Edmonton, Alberta,
Canada T6J 4M2

Date: . . . .

University of Alberta

An Assessment of Java/RMI for Object Oriented Parallelism

by

Alberto Daniel Zubiri

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1997

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Assessment of Java/RMI for Object Oriented Parallelism** submitted by Alberto Daniel Zubiri in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . .
Dr. Jonathan Schaeffer

. . . .
Dr. Ron Unrau

. . . .
Dr. Duane Szafron

. . . .
Dr. Xiaoling Sun

Date: . . . .

To my Family

Abstract

Java is a relatively new object-oriented language that supports concurrency and network programming; two characteristics that make it an ideal candidate for the development of parallel applications using networks of workstations. The latest version of the language incorporates a new form of communication between objects called *Remote Method Invocation (RMI)*. This feature allows objects to transparently invoke methods across the network.

This thesis explores the issues involved in automatically transforming sequential object-oriented programs written in Java into equivalent distributed programs that will run on a network of workstations and whose objects will transparently communicate by using the RMI system. The focus is placed on how the features of the language, the object-oriented technology used, and the parallelizations techniques applied can be combined for the development of such distributed programs.

This thesis uncovers some hard problems that will need further research. Exceptions, a feature that is common in many of today's languages (e.g. C++), can seriously limit the amount of concurrency obtained by the use of asynchronous messages. The dynamic dispatching feature, common in many object-oriented languages, can become a problem for properly identifying the objects that need to be changed when some other objects are placed in a different address space. This thesis identifies many of the problems introduced by RMI. Solutions for most of these problems are presented. However, given its performance, we conclude that the amount of compiler support needed to address all these problems is too large compared with the benefits obtained by using the RMI system as the underlying communication paradigm of our parallelizing tool.

Acknowledgements

First and foremost, I thank my wife, Adriana, for her love and care. Her unconditional support gives me the confidence and strength necessary to tackle any task. I also thank my family in Argentina for all their understanding and encouragement. Specially I want to thank my mom, Maria Eugenia, and my dad, Lito, for giving me the life and for always letting me choose my own way. Thanks also to my family-in-law, Coco, Zuny and Ceci for the the thousands of e-mails that helped me to cope with the distance.

I also want to take this opportunity to thank my supervisors, Dr. Jonathan Schaeffer and Dr. Ron Unrau, for their guidance and funding. The amount of time and effort they dedicated to me greatly improved my work. I am also grateful to Dr. Duane Szafron and Dr. Xiaoling Sun for taking the time to read and comment on this thesis, and to Dr. Joe Culberson who chaired my thesis defense.

Help from my colleagues in the Parallel Programming Systems Research Group, is highly appreciated. I would also like to thank to Diego and Liliana Novillo, and to Andreas and Manuela Junghanns for all their support, company and friendship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays, the computing world is changing faster than ever; new computer systems are deployed every day and the explosion of the World-Wide Web is changing the way people do business, interact and communicate. Networks of workstations are common in many organizations. Moreover, as the price of computer hardware drops, the computing power available in today's organizations continues to grow. As the availability of low-cost, high-performance workstations continues to increase, there is also a growing number of unused CPU cycles. Many efforts have been made to take advantage of this potential computing power by combining networks of workstations to form powerful supercomputers. In these networks, parallel applications can be developed to solve larger problems than before and to obtain increased performance.

However, parallel programming is a difficult task that involves dealing with many issues not present in sequential programming such as communication, synchronization, non-deterministic program behavior and work distribution [SS96]. Furthermore, if the underlying parallel computer is a network of workstations, issues such as network partitioning and network latency also need to be addressed.

A broad spectrum of parallel programming tools have been developed to cope with the complexities of parallel programming. Those seeking the maximum performance prefer low-level code libraries that have a minimal amount of overhead. Examples of these low-level tools include sockets and message passing libraries (such as PVM [Sun90] and MPI [MPI94]). Programming with these libraries is a highly skilled task. The programmer is free to choose the best way of parallelizing the application, but is also responsible for explicitly handling the communication, synchronization, and many more low-level issues.

Many attempts have been made to alleviate the intrinsic complexity of parallel programming mainly in the form of high-level parallel programming tools and environments that support parallel extensions to existing languages or new parallel languages. Examples of these environments are Enterprise [SS+93] and HeNCE [BD+93]. Parallel programming environments allow the programmer to concentrate on the algorithm's development rather than on the low-level details of the parallelization (e.g. processor allocation, communication protocols, etc.) by providing the programmer with an abstract model. The main disadvantage of these parallel programming environments is the lack of flexibility when the structure imposed by their models differ from that of the application.

Since object-oriented technologies have been successfully applied to develop better sequential applications, it is possible that those technologies would benefit the parallel applications' world. Object-oriented languages have been successfully used to implement powerful and flexible software systems. In the object-oriented paradigm, an object is a self-contained entity which has a private structure and a public interface. An application implemented under this paradigm can be seen as a collection of communicating objects which cooperate to obtain the desired result. These objects communicate by exchanging messages according to their interfaces.

In the case of sequential execution, all the messages are exchanged in a fixed execution order which is directly derived from the statement order in the program source. There is exactly one thread of control in which objects communicate. The communication is performed synchronously while the thread of control is passed together with each message sent from one object to another. However, if messages are sent asynchronously without the accompanying thread of control, then the sender may continue executing concurrently with the receiver of the message. This leads to a model in which a program will be executed by several threads of control. Under this model, an application is represented as a collection of concurrently executing objects that communicate using message passing.

Traditional object-oriented languages such as Smalltalk [GR83], Objective C [Cox86] and C++ [Str91] have been developed with special emphasis on software engineering concepts such as reusability, encapsulation, inheritance and polymorphism. Little or no attention has been paid to concurrency.

However, the idea of applying object-oriented techniques to the programming of concurrent applications has gained popularity in recent years. Several object-oriented languages have been created to deal with concurrency (e.g. Actors [AH87], POOL-T [Ame87], HYBRID [Nie87]) and many existing languages have been extended to support it (e.g. ConcurrentSmalltalk [YT87], μC++ [BD+92]). In particular, among the new languages recently created, Java [GJS96] supports concurrency and network programming which makes it an ideal candidate for the development of parallel applications using networks of workstations. Furthermore, the latest version of the language incorporates a new form of communication called *Remote Method Invocation (RMI)* [RMI97]. This feature allows objects to transparently invoke methods across the network. Ideally, this will facilitate the development of object-oriented parallel applications that will have their objects distributed throughout the network. Those objects will transparently cooperate regardless of their location by exchanging messages, and a high degree of parallelism will be attained by using the processing power of the participating nodes.

1.2 Scope of the Thesis

This thesis addresses the issues involved in automatically distributing sequential object-oriented programs written in Java by using a parallelizing tool (e.g. parallelizing compiler and/or programming environment). The goal of such a tool is to translate sequential Java programs into equivalent distributed programs that will run on a network of workstations and whose objects will transparently communicate by using the RMI system. In particular, the focus is placed on how the features of the language, the object-oriented technology used, and the parallelization techniques applied can be combined for the development of such distributed programs.

While doing this work, several problems were encountered that had a significant impact on how to generate object-oriented parallel applications. Each of the problems encountered is analyzed to determine its source, and possible solutions are explored. While some problems are directly related to the Java/RMI design, others are intrinsic to the object-oriented characteristics of the programs (e.g. problems due to inheritance and dynamic dispatching); or to parallel computing (e.g. non-deterministic program behavior).

This thesis uncovers some hard problems that will need further research. Namely, it is shown how the use of exceptions, a feature that is common in many of today's languages (e.g. C++), can seriously limit the amount of concurrency obtained by the use of asynchronous messages. Also, it shows how the dynamic dispatching feature, common in many object-oriented languages, can become a problem for properly identifying the objects that need to be changed when some other objects are placed in a different address space. RMI, in addition, presents many problems that can be addressed as discussed in this thesis. However, given its performance, we also conclude that the amount of compiler support needed to address all these problems is too large compared with the benefits obtained by using the RMI system as the underlying communication paradigm of our parallelizing tool.

Other contributions of this thesis include: a performance evaluation of the RMI system, solutions to some RMI-related problems, and a prototype runtime system for distributing Java programs.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 introduces the Java language, its design goals and its main features. Chapter 3 presents Remote Method Invocation (RMI). Its main characteristics are discussed and an example is provided. In Chapter 4, object serialization is explained in detail. Chapter 5 presents a performance evaluation of RMI and object serialization. Chapter 6 shows how inheritance fits with the RMI system. The concept of futures is explained in Chapter 7 with a discussion of different alternatives for implementing them in Java. Chapter 8 covers the topic of collaborators. A prototype runtime system is described in Chapter 9 and related work is presented in Chapter 10. Finally the conclusions of this thesis are presented in Chapter 11.

Chapter 2

The Java Language

The Java programming language has its roots in a previous language called Oak [Eng97]. Oak was the main result of a research project started at Sun Microsystems Inc. around 1990 that targeted the development of software for embedded systems and consumer electronics devices. This project started with C++ as the programming language. However, after many problems with C++, the development team decided that they were better off creating an entirely new language and Oak was born.

After the introduction of Mosaic and the World Wide Web in 1993, the research project shifted direction and focused on the development of software systems for online multimedia. Oak was positioned as a "language-based operating system." Its source code was given away for free on the Internet. Before releasing the software on the Internet and after considering different names (such as Neon, Lyric, Pepper and Silk) the name was changed to **Java**.

Then Netscape, which quickly became the most popular Web browser, made a commitment to support Java. After that, Java has become the preferred language of the Internet community, and software written in Java can be seen everywhere.

2.1 Design Goals

Java's origins as a language for programming consumer electronics, and the problems faced with C++ in the beginning, contributed to set the following design goals for Java [JO95, GM96]:

Simple, Object Oriented, and Familiar: The new language should be easy to learn so that programmers could become productive with little training. Even though C++ proved to be unsuitable, Java was designed to look as close as possible to it. This allowed C++ programmers to look at Java as something *familiar* and be productive in the language very quickly. To keep the language simple though, many complex features of C++ were omitted, such as operator overloading, multiple inheritance, and extensive automatic coercions. Automatic *garbage collection* was also included to simplify the programming and to avoid bugs due to storage management errors that were common in C++. The language was made *object oriented* borrowing characteristics from C++, Objective C, SmallTalk, Eiffel, and Cedar/Mesa.

Network-savvy: The Java language provides direct support for network programming with libraries that simplify the complexities of dealing with different protocols such as TCP/IP, FTP and HTTP.

Robust and Secure: Software embedded in consumer electronics requires the highest level of robustness. Java attains this with extensive compile-time and run-time checkings and with language features that enforce healthy programming practices such as exception handling (see Figure 2.1). There are no pointers and the dynamically allocated memory is automatically garbage collected. This characteristic makes the software more robust and secure since applications are not allowed to manipulate addresses to access data that they are not supposed to. Also, the runtime system provides extra validations to ensure that Java systems are secure from external tampering.

Architecture Neutral and Portable: The need for running programs on different hardware architectures (i.e. different consumer electronics devices or different computers in an heterogeneous network), called for an *architecture neutral* design. Java programs are compiled to an architecture-independent format, called *bytecodes*, which can be run on any Java-enabled platform without modification (this idea was borrowed from Smalltalk [GR83]). Java goes one step further by defining the sizes of the primitive data types and also the behavior of arithmetic on them (e.g. floats are always 32-bit IEEE 754 floating point numbers).

Interpreted: To run the same binary program on any hardware platform without

```
try {
    // Code that may raise exceptions (via the throw statement).
    // Also, the block may be exited by a break, continue or return
    // statement.
}
catch(exception_name e) {
   // code that will handle the exceptions of type exception_name (or one
   // of its subtypes.
}
finally {
   // This code is always excecuted regardless how the try block was
   // exited. That is, after the end of the try block, or a break, continue
   // or return statement is reached, or after an exception is handled by a
   // catch block or after an exception is raised and not handled.
}
```

Figure 2.1: Java's Exception Handling Mechanism

recompiling, the bytecodes are interpreted and translated on the fly to the particular machine code suitable for the hardware the application is running on.

High Performance: Performance is a key design issue to any software system. Even though Java is an interpreted language, high performance is still attainable by exploiting multithreading, native methods and JIT compilers. [1]

Multithreaded: Many applications today require the ability to have several things going on at the same time. For example, a Web browser could be downloading a file and at the same time displaying a page with sounds and live video. Each of these concurrent activities can be executed by a separate thread. Threads are part of the Java language and multithreading is built into every component of the runtime system (e.g. the garbage collector runs in a separate low-priority thread). System libraries are *thread-safe* and the language provides powerful thread synchronization primitives.

Dynamic: Java is dynamic in the sense that it adapts to an evolving environment.

[1] Just in time (JIT) compilers translate the bytecodes into machine code on the fly for the platform where the application is running. If the same method is called again, the compiled version is used instead, thus improving performance.

Classes are loaded from different sources (even from the network) and linked only when they are needed. Any object in Java can be queried about its class and new instances of a class whose name is obtained at runtime can be dynamically created.

2.2 Language Features

The Java language features follow, to some extent, those found in the C/C++ languages [GM96]. However, some of the C/C++ features are not present in Java. These include: typedefs, preprocessor, structures, unions, enums, free-standing functions, multiple inheritance, goto statements, operator overloading, automatic coercions, and pointers. These features were omitted by the language designers in their quest for an easy to use and robust language (e.g. automatic coercions can be the source of bugs when the programmer is not fully aware of them).

The basic features incorporated into the language include primitive data types (e.g. `boolean`, `int`, `double`, etc.), arithmetic and relational operators, scope modifiers (e.g. `private`, `public`, `protected`, etc.), arrays, strings, control flow and decision-making statements, and an exception handling mechanism. Among the advanced features included in the language are:

Object-Oriented Features: Java is a truly object-oriented language in the sense that it supports objects, classes and inheritance as defined by Wegner [Weg87]. It also supports encapsulation, polymorphism, and dynamic binding. In Java, every class has exactly one parent (i.e. single inheritance) and has a common ancestor (i.e. `Object`). Classes are also objects in Java. Parameters and return values are always passed by reference. Abstract classes, interfaces and visibility modifiers (e.g. private, protected, etc.) are also supported. See [GM96] for a complete reference on these topics.

Memory Management and Garbage Collection: Memory management is very simple in Java. New memory is allocated with the `new` operator but, unlike in C/C++, there is no `delete` operator. Dynamically allocated memory is automatically garbage collected. The garbage collector runs in its own low-priority thread, freeing storage and compacting the memory pool.

Integrated Thread Synchronization: The Java language provides direct support for threads. There is a Thread class and the language provides powerful synchronization primitives.

Interfaces: Since Java only supports single inheritance, an interface feature is provided. Interfaces allow Java to create the illusion of multiple inheritance. An interface is a contract that a class agrees to implement. Saying that a class *implements* a particular interface means that it provides an implementation for all the methods declared in the interface. Moreover, interfaces are data types in Java, which means that instances of a class that implements a particular interface can be assigned to variables of the interface type. Interfaces are used to specify functionality without specifying a specific implementation. A class can implement any number of interfaces and inheritance can be applied to them. An interface can extend one or more interfaces, that is, interfaces support multiple inheritance.

Packages : A package is a collection of classes that are related in some way. Every Java class belongs to a package. Classes for which a package is not specified belong to the default package. The concept of package allows a programmer to separate the classes into different modules. Java also provides scope modifiers which can regulate the access based on the class' package.

Security: Security played a big role in the definition of the Java language and its runtime system. Any application developed in Java is subject to four levels of security [Ham96]:

1. **Language and compiler:** The lack of pointers in the language prevents attacks made by the manipulation of addresses. Since the compiler does not allocate memory, a potential hacker cannot infer any physical layout by looking at a declaration. Moreover, there is no language construct that permits a programmer to obtain the address of an object and/or method.

2. **Bytecode verifier:** To prevent attacks from hostile Java compilers, the runtime system subjects all code to a simple theorem prover that performs some validations (e.g. it checks whether the code violates access restrictions and whether all the instruction parameters have the correct type).

3. **ClassLoader:** Each class that is loaded from the network executes within its own separate name space. All the classes loaded from the local file system share the same name space. When a class references another, the referenced class is looked up first in the local name space. If not found, then it is looked up in the name space of the referencing class. This prevents code loaded from the network from "shadowing" a built-in class or a class loaded from a different site.

4. **Interface specific security:** The Java networking package can be configured to provide different levels of security for the protocols that it interfaces to (e.g. HTTP, FTP, etc.).

2.3 The Java Platform

Nowadays, many different and usually incompatible platforms exist (e.g. Microsoft Windows, Macintosh, OS/2, UNIX, etc.). Generally, applications developed for one platform do not run on another (at least not without considerable effort).

Java introduces a new software platform that sits on top of any existing platform [Kra96]. This allows the deployment of applications that will run on any machine where the Java platform is implemented, without recompiling the source. Even though there are many implementations of the Java platform (i.e. one for each different hardware/OS), there is only one specification for this platform. This makes it possible to develop the applications just once and run them everywhere. Two kinds of programs can be deployed on the Java Platform:

Applets: Java programs that need a browser to run. These program are downloaded from the network and run on the client machine.

Applications: Stand-alone Java programs. These programs do not require a browser to run and do not have a built-in downloading mechanism.

The Java Platform has two main components:

Java Virtual Machine: The Java Virtual Machine is an abstract computer that can be implemented (either in software or hardware) on top of any real computer.

Java Application Programming Interface (Java API): The Java API provides a standard interface to applets and applications to the Java Virtual Machine that is the same across all the operating systems. The Java API can be further divided into:

- **The Java Base API:** Which provides the basic language, utilities, I/O, network, GUI, and applet services.
- **The Java Standard Extension API:** Which provides a framework to incorporate capabilities that are beyond the Java Base API (e.g. Java Enterprise, Java Commerce).

2.4 Summary

This chapter introduced the Java language. Its design goals and main features were explained. The Java platform was also introduced. Java is an interpreted language that among other features supports: tree single inheritance, interfaces (with multiple inheritance), dynamic dispatching, different scopes, threads and exception handling. As we will see in the following chapters, some of these features will present special challenges when trying to automatically parallelize sequential Java programs.

Chapter 3

Remote Method Invocation

3.1 Introduction

Distributed systems require the existence of some form of communication between applications running in different address spaces. Java supports sockets as a general form of communication between objects running on different virtual machines (which could be running on the same or different hosts). Socket programming, however, is a demanding task since all the details of the communication have to be handled by the programmer. Before two applications can communicate using sockets, the developer has to design protocols (i.e. which communication pattern the interactions will follow) and message formats (i.e. the structure of the transmitted data). This is a time-consuming and error-prone activity. Furthermore, once in place, protocols and message formats are hard to change so the system becomes more difficult to maintain.

Starting with version 1.1, Java provided a new type of communication between objects called *Remote Method Invocation (RMI)* [RMI97]. This new abstraction permits the developer to concentrate on the application development rather than the complexities of the communication process.

3.2 The RMI System

The RMI system is built on top of the socket mechanism and provides the semantics of RPC (Remote Procedure Call) [BN83] applied to distributed object-oriented systems. Even though the design of the RMI system is general enough to accommodate asynchronous messages, the current implementation only supports synchronous method calls. In the RMI model, a *Remote Object* is defined as one that implements some methods that can be called from a different Java Virtual Machine. [1] A remote method invocation is then the act of calling a method of a remote object defined in a remote interface implemented by this object. The syntax for invoking one of these remote methods remains the same as the one used for a local call. This characteristic makes RMI easy to use because there is no new syntax to learn, although the user still must be familiar with the concepts of remote objects and remote method invocations. The methods exported by a remote object are described by one or more remote interfaces. Clients will see only what is defined in the remote interface of a remote object. This implies that direct access to instance variables is not possible with RMI (only methods and constants can be defined in a Java interface). Also, it implies that RMI cannot be used with static methods because the modifier static is not allowed in a method declaration inside an interface. Furthermore, since all the methods in an interface are implicitly abstract and public (i.e. the modifiers private and protected are not allowed inside interfaces), all the methods that will be called remotely need to be public.

References to remote objects can be returned as method return values or used as parameters in any method invocation, no matter whether the object that implements the method is local or remote. Any parameter or returned result in a remote method invocation that is not a remote object is passed by value. Remote objects are passed by reference.

For a client to invoke a remote method, it first has to get a reference to the remote object. The RMI system provides a simple name service, called the *registry*, that can be used for this purpose. The registry is a "well-known" remote object whose responsibility is to map names to remote objects. A common configuration consists of a single registry for each host, but it is also possible to have a registry for each server process (either in the same or in a different virtual machine). Clients can access the registry directly using the methods defined in the Registry interface and the LocateRegistry class, or using the URL-based methods provided by the class java.rmi.Naming. The Registry interface provides methods for lookup, binding, un-

[1] Note that a remote object can still have local methods that can only be called from objects that reside on the same virtual machine as the remote object.

binding, rebinding and listing the contents of the registry. The LocateRegistry class provides static methods that allow a program to retrieve a reference to a registry given different parameters such as the host and port number.

Since there are more opportunities for failures in distributed systems (due to network partitioning, routers dropping packets, etc.) than in local systems, clients of a remote method invocation must be able to handle additional exceptions thrown by the RMI system. All these exceptions are subclasses of java.rmi.RemoteException. Thus, every method declared in a remote interface must add java.rmi.RemoteException to its throws clause. When an error occurs, the RMI system throws a RemoteException. However, the client may have little or no information for determining if the failure happened before, during, or after the remote call.

3.3 RMI Architecture

The RMI system is composed of three independent layers: the stub/skeleton layer, the remote reference layer, and the transport layer (see Figure 3.1). Each layer communicates directly only with the layer immediately above or below it. For example, user programs (i.e. client and servers) only talk directly with the stub/skeleton layer. The boundaries between any two layers are defined by precise interfaces and protocols. This makes each layer independent of the next and, thus, replaceable by another implementation that adheres to the proper interface and protocol. For example, the current implementation of the transport layer is based on TCP sockets but this can be replaced by an implementation based on UDP sockets without affecting the other layers.

Figure 3.1 illustrates a client communicating with a server. Both objects reside on different virtual machines and the client invokes a remote method implemented by the server. The stubs and skeletons are automatically generated by the RMI compiler (rmic) from the remote interfaces. Clients interact with stub objects (proxies of the remote object) that have exactly the same set of remote interfaces defined by the remote object's class. Non-remote portions of the class hierarchy of the remote class are not included in the stub. Thus, only those methods declared in the remote interfaces can be called remotely.

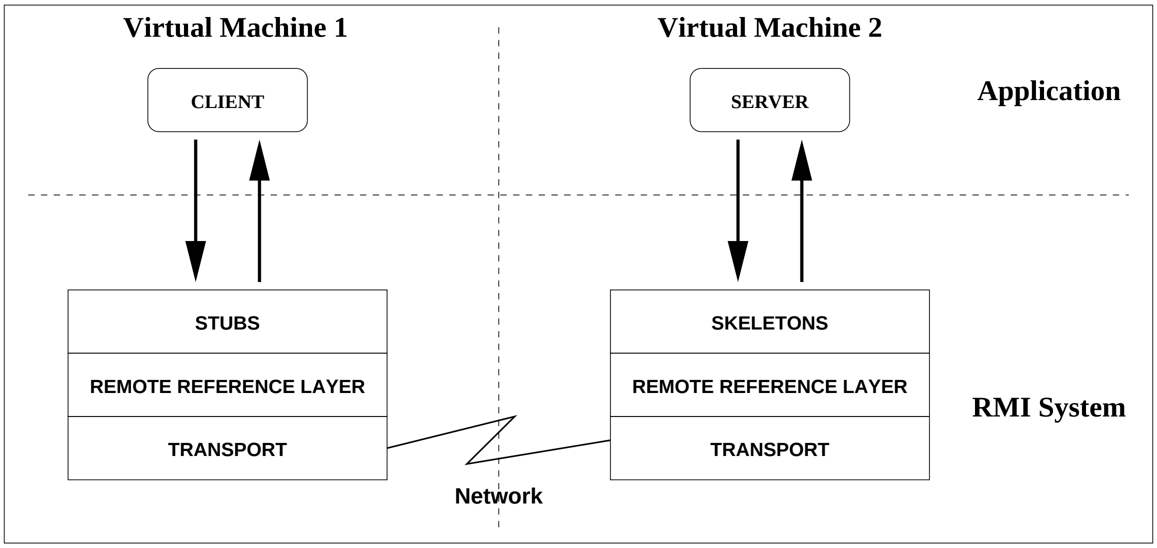

Figure 3.1: RMI Architecture

A remote call made by the client is handled by the stub which relays all the invocations to the server via the reference layer. The remote reference layer is used to abstract different semantics for an invocation (e.g. unicasting, multicasting, etc.). The reference layer uses the services of the transport layer for setting up connections and remote object tracking. The skeleton at the server side receives the invocations from the transport layer and makes the appropriate calls to the remote object which executes the actual method calls. The return values are sent back to the client from the server to the skeleton, then to the remote reference layer, from there to the transport layer, up to the stub until finally reaching the caller.

The transmission of objects between different address spaces is done using Java's Object Serialization API which provides the functionality necessary to pack and unpack any arbitrary graph of objects (see Chapter 4 for a more complete description).

3.4 Example

To illustrate how RMI is used, consider the code in Figure 3.2. There are two classes: Sensor, which represents a physical sensor of some sort; and MAIN, which is used to provide an entry point for the program. [2] The program consists of creating an object

[2]In Java, all the programs begin executing the code contained in the main() method of the specified class (i.e. MAIN in our example).

```
//—[ Sensor.java ]——————————
public class Sensor {
  private boolean active;

  public Sensor() {                              // Constructor
      active = false;
  }
  public void Activate() {                 // Callable method
      ...
      active = true;
  }
  public void Deactivate() {               // Callable method
      ...
      active = false;
  }
}

//—[ MAIN.java ]——————————-
public class MAIN {
    public static void main(String args[]) {
        // Create and initialize a new Sensor object
        Sensor aSensor = new Sensor();
        // Access it
        aSensor.Activate();
        ....
        aSensor.Deactivate();
    }
}
```

Figure 3.2: RMI Example: Local Code

of type Sensor, called aSensor, and operating with it. The sensor is activated using the method Activate() and, after doing something with it, the sensor is deactivated by Deactivate(). The code shown in Figure 3.2 runs locally in a single virtual machine.

Figures 3.3 and 3.4 show the same program translated into a distributed version using RMI. The original program has been split into a server which provides the functionality of the sensor, and a client which uses the sensor. In this version, the client and the server run on different virtual machines.

Both server and client main() functions start by setting the system's security manager to an instance of RMISecurityManager. Since the code for the stubs and skeletons

```
//—[ SensorRI.java ]——————————————————————
public interface SensorRI extends Remote {
    public void Activate() throws RemoteException;
    public void Deactivate() throws RemoteException;
}

//—[ Sensor.java ]——————————————————————
public class Sensor extends UnicastRemoteObject implements SensorRI {
   private boolean active;

   public Sensor() throws RemoteException {
       active = false;
   }
   public void Activate() throws RemoteException {
       ...
       active = true;
   }
   public void Deactivate() throws RemoteException {
       ...
       active = false;
   }
   public static void main(String args[]) {
      System.setSecurityManager(new RMISecurityManager());
      try {
           Registry registry = LocateRegistry.getRegistry();
           Sensor aSensor = new Sensor();
           registry.rebind("SENSOR", aSensor);
      } catch (Exception e) {
           System.out.println("Sensor.main: an exception has occurred: " +
                                  e.getMessage());
           e.printStackTrace();
           System.exit(-1);
      }
      System.out.println("Remote Sensor ready.");
   }
}
```

Figure 3.3: RMI Example: Server Code

```
//—[ MAIN.java ]——————————————————————-
public class MAIN {
    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            Registry registry = LocateRegistry.getRegistry("serverHostName");
            SensorRI aSensor = (SensorRI) registry.lookup("SENSOR");

            aSensor.Activate();
            ...
            aSensor.Deactivate();
        }
        catch (Throwable e) {
            System.out.println("MAIN: an exception has ocurred: "+e.getMessage());
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Figure 3.4: RMI Example: Client Code

can potentially be loaded from the network, there must be a security manager in place or an exception will be thrown. This security manager will ensure that the loaded classes (i.e. the bytecodes that were brought into the virtual machine from the network or from disk) adhere to the standard Java safety guarantees. Applications can define their own security managers or use the provided RMISecurityManager. If no security manager is in place, no class can be loaded from the network.

The code for the server (see Figure 3.3) is divided into two files: SensorRI.java and Sensor.Java. SensorRI is an interface used to declare which methods can be remotely called by a client. The class Sensor implements the SensorRI interface. The main() function in the server code just creates an instance of sensor and registers it with the local registry.

The code for the client (see Figure 3.4) shows how to remotely invoke methods on the remote Sensor object. First an object reference is obtained using the registry. The registry contacted is the one that it is running on the host where the server has been started. If the remote object (i.e. the server) has not yet been created and bound to the registry, the registry.lookup() call will throw an exception (i.e. java.rmi.NotBoundException). The reference obtained from the registry has the type of the interface SensorRI rather than the class Sensor. This is because only those things declared in the remote interface are seen from a different virtual machine. Once a reference has been obtained, the methods can be invoked using the normal Java syntax. The calls are placed inside a try{} clause in order to deal with the remote exceptions that can be thrown.

3.5 Discussion

Having explained the basic functionality of RMI, it is important to discuss the different issues involved in using it for the automatic parallelization of sequential code. A parallelizing compiler at a bare minimum should automate the generation of the remote interfaces. Given a class (through a modifier or using a graphical interface), the compiler should be able to construct the remote interface by listing all the methods of that class and adding the appropriate throws clauses for the remote exceptions. Also, RMI requires all the methods that will be called remotely (i.e. those that are added to the remote interface) to be declared as public. Thus, the compiler should verify (and change if necessary) the access privileges for each method that is added to the remote interface. The compiler should also insert the appropriate try and catch statements around each remote method invocation to automatically handle all the exceptions thrown by the RMI system.

Other issues where compiler support will be needed include: modifications to the user code for handling access to instance and class variables, packing optimizations, dynamic object creation, access to static methods, and asynchronous method calls. These and other issues derived from them will be addressed in detail in subsequent chapters.

Chapter 4

Object Serialization

In distributed systems, data structures must be flattened before transmission (i.e. they are put in a buffer for transmission according to a particular layout). At the receiving end, the data structures are reconstructed from the flattened version. This characteristic is not exclusive to object-oriented systems. In fact, the packing of data structures also has to be performed in procedural languages (such as C) when performing remote procedure calls (RPC) [BN83] or saving these structures to disk. This process is known as "packing", "pickling", and "marshaling". In Java it is called *Object Serialization* although, besides objects, it is also used to pack primitive data types.

Serialization is an important issue for parallelizing applications. When an object is to be sent across the network, its state must be saved at one end and restored at the other end. If the object contains references to other objects, there are two possibilities: to pack the object and all the objects referenced from it (deep copy), or to pack only the original object (shallow copy). Clearly, using shallow copies is faster than using deep copies, but sometimes a deep copy is needed to preserve program correctness (e.g. if the remote method accesses a nested reference, the referenced object should also be packed).

This chapter explains how object serialization is performed in Java. The focus is placed on analyzing how the serialization mechanisms affect the automatic parallelization process.

```
FileOutputStream fos = new FileOutputStream("some_file");
ObjectOutputStream os = new ObjectOutputStream(fos);
// writing of an object
os.writeObject("This is a String object");
// writing of a primitive data type
os.writeInt(100);
os.flush();
```

Figure 4.1: Writing to an Object Stream

4.1 Object Serialization in Java

Object serialization provides the capability of storing and retrieving objects from a stream. This means that the objects can be made persistent (i.e. written to disk) and, also, that the objects can be transmitted through a communication channel (e.g. sockets). Thus, to *serialize* an object consists of writing to the stream all the information needed to create an equivalent copy of the object at a later time in the same or a different place. The process of creating an equivalent object from the stream is called *deserialization*. When serializing an object, all the objects that are being referenced by it are also serialized to the stream in order to maintain the existing relationships among them (i.e. deep copy).

The programmer can specify if a particular class is serializable or not (i.e. if its objects can be written to a stream). For example, a programmer might not want to make a class serializable for security reasons. Classes whose supertype is not serializable may still be serializable but have to take responsibility for packing and unpacking its parent's public, protected and package protected fields. This is only possible though, if the supertype has a constructor with no arguments (which will be used to initialize the supertype's fields).

Normally, objects are written to the stream using the writeObject() method declared in the ObjectOutput interface. Primitive data types are written using specific methods declared in the DataOutput interface such as writeInt(), writeFloat(), etc. Figure 4.1 shows how a String object and an int are serialized.

The writeObject() method handles the serialization of each object and all the objects referenced from it. The object, all the objects that are referenced from it,

```
FileInputStream fis = new FileInputStream("some_file");
ObjectInputStream is = new ObjectInputStream(fis);
// reading an object
String aString = (String) is.readObject();
// reading a primitive data type
int i = is.redInt();
```

Figure 4.2: Reading from an Object Stream

and recursively any object that is being referenced from them, form the object's graph. Objects are serialized at most once in a stream. If more references to the same object are found while traversing the graph (i.e. when following the references) they are encoded as a handle (i.e. an offset inside the stream) to the already serialized object. This allows for a very efficient representation which supports the cycles that can be present in an arbitrary object graph. Fields that are marked as transient or static are not serialized/deserialized. This is because transient fields are not part of the object's state by definition (i.e. the keyword is used for that purpose), and static fields are class variables (i.e. not part of the object's state but of its class).

Reading from a stream is also a straightforward process (see Figure 4.2). The method readObject() is used to deserialize any arbitrary graph of objects and the methods readInt(), readFloat(), etc. are used to deserialize the primitive data types.

The *Object Serialization API* requires that classes implement either the Serializable or the Externalizable interface in order to serialize the objects of that class. For classes that implement Serializable, all the information required to restore its objects is automatically saved. In contrast, for classes that implement the Externalizable interface, only the identity of the class is automatically saved and the class is responsible for saving and restoring its contents (by implementing the writeExternal() and readExternal() methods).

The Serializable interface is an empty interface (i.e. there is no functionality to implement) and it is used for the sole purpose of identifying a class as serializable. In the early versions of the API, all classes were serializable by default. Later on, this default behavior was changed and only those classes explicitly declared to implement

Serializable could have their state automatically saved to a stream. [1] The rationale behind this change in the design was twofold. The first consideration had to do with security. Access to fields that are private, protected or package protected is regulated by the Java runtime system. Once serialized, however, the byte stream can be accessed and changed by any object that has access to it. This violates the privacy guarantees of the language and could compromise the integrity of the Java environment. Making classes not serializable by default prevents a programmer that does not know about serialization from compromising the security and/or the integrity of the system because of his/her lack of knowledge.

The second reason for the change in the design is to force the designer to think about serialization before tagging a class as serializable. For example some fields make sense only in the context where the object containing them was originally created (e.g. open sockets, open files, etc.). These fields should be marked as transient but the programmer could neglect or forget to do so. In that case, if the classes were serializable by default the serialization process could introduce incorrect behavior into the system.

4.2 The Serialization Process

The ObjectOutputStream class implements the core of the serialization process. This class is responsible for keeping the state of the stream in which the objects are written and also the state of the objects already serialized.

This class provides methods for writing different data types to the stream (e.g. writeInt(), writeLong(), etc.), and also for manipulating the stream (e.g. flush(), drain(), close(), etc.). Other methods provided by this class allow the traversal of the objects that need to be serialized. Objects are saved on the stream together with the objects to which they refer (i.e. complete object graphs are saved). In particular the writeObject() method is the one that implements the serialization of an object into a stream. Objects are serialized according to the following algorithm [OS96]:

1. If the block-data buffer [2] is not empty, its contents are written to the stream

[1] An extensive discussion about this issue can be found in the archive for the RMI mailing list located at: http://chatsubo.javasoft.com/email/rmi-users/.

[2] The stream can be buffered or unbuffered. When using buffered mode, the data is put in the

and the buffer is reset.

2. If the object being serialized is null, writeObject() returns after putting null on the stream.

3. If the object was previously written to the stream, its handle is written instead. If the object had been replaced, the handle for the replacement is written instead. Then writeObject() returns. Note that the serialization routines are forced to keep track of which objects have already been packed and which have been replaced.

4. If the object being written is a Class, the corresponding ObjectStreamClass is written, a handle is assigned for the class and writeObject() returns. An ObjectStreamClass describes a class that can be serialized to a stream or deserialized from it. The ObjectStreamClass for a loaded class can be found using a lookup method provided by ObjectStreamClass.

5. If the object is an ObjectStreamClass, a class descriptor that includes name, serialVersionUID [3], and the lists of fields ordered by name and type is written to the stream. A handle is assigned for this descriptor and the annotateClass() method is called before writeObject() returns.

6. If the object is an instance of java.lang.String, the object is written in Universal Transfer Format (UTF) [4] format and writeObject() returns after assigning a handle to the string.

7. If the object is an array, a recursive call to writeObject() is used to write the ObjectStreamClass of the array. A handle is assigned for the array, and then the length is written to the stream followed by each element of the array. Finally, writeObject() returns.

8. If object replacement is enabled by a previous call to enableReplaceObject(), the replaceObject() method is called to allow subclasses to substitute an object. If

block-data buffer before being written out to the stream.

[3] The serialVersionUID is a number that identifies a particular version of a serialized class.

[4] UTF is a "multi-byte" encoding format used for the storage and transmition of Unicode characters (see [Fla96, 208-209]).

the object is effectively substituted, the mapping between the original object and its replacement is saved for later use in step 3. After the replacement, steps 2 through 7 are repeated for the new object. If the replacement object does not correspond to the types handled in steps 2 through 7, the processing continues at step 9 with this new object.

9. For regular objects a recursive call to writeObject() is used to write the ObjectStreamClass of the object's class. A handle is created for this object.

10. Finally, the contents of the object are written to the stream according to the following criteria:

The object is Serializable: The highest Serializable class is located, and for this class and each derived class its fields are written. If the class implements a writeObject() method, it is called. If not, the defaultWriteObject() method is called instead. This method writes all the non-static and non-transient fields to the stream. Note that writeObject() can use defaultWriteObject() to write the state of the object and then write additional information.

The object is Externalizable: In this case the writeExternal() method is called. This method should save the entire state of the object and coordinate with its superclasses to save their state.

The object is neither Serializable nor Externalizable: When the object is not Serializable or Externalizable, a NotSerializableException is thrown.

Even though it is not necessary to know the complete details of the serialization process in order to use it, some general understanding will help to better utilize it. Specifically, it is important to know the difference between Serializable objects and Externalizable ones. Also note that this is a generic serialization routine and, thus, it is very complex. Users could get better performance by writing their own custom serialization code.

4.3 The Deserialization Process

Object deserialization is implemented by the ObjectInputStream class. This class is responsible for maintaining the state of the stream and the objects already deserialized. It provides methods for reading objects and primitive data types written by ObjectOutputStream. The readObject() method is the one that implements the object deserialization from the stream. Objects are deserialized from the stream according to the following algorithm [OS96]:

1. If a block-data record is present on the stream, a BlockDataException is thrown with the number of bytes that are available for reading.

2. If the object in the stream is null, null is returned.

3. If the object in the stream is a handle to a previous object, the appropriate object is returned from the set of known objects.

4. If the object is an instance of java.lang.String, then its UTF encoding is read and the object and its handle are added to the set of known objects. Then the String is return.

5. If the object being read is a Class, the corresponding ObjectStreamClass descriptor is read. The Class object is returned after it is added with its handler to the set of known objects.

6. If the object is an ObjectStreamClass, its name, serialVersionUID, and fields are read. Then, the object and its handle are added to the list of known objects. Following that, the method resolveClass() is used to get a local class for this descriptor (an exception is thrown if the class cannot be found). Finally, the ObjectStreamClass object is returned.

7. If the object is an array, its ObjectStreamClass and length are read. The array is allocated and added with its handle to the set of known objects. Then, each element is read (according to its type) and assigned to the array. Finally the array is returned.

8. For any other object, its ObjectStreamClass is read from the stream and used to retrieve a local class. The class has to be Serializable or Externalizable.

9. The class is instantiated, and the instance and its handle are added to the set of known objects. Then, the contents are retrieved according to the following criteria:

   Serializable objects: the no-argument constructor for the non-serializable supertype is called and each field is restored using the readObject() method or the defaultReadObject() method if necessary. If the version of the class that wrote the stream and the version of the class that is reading the stream differ, it could happen that both have different supertypes. If this is the case, the ObjectInputStream must match the available data with the classes that are being restored. Data for classes that are in the stream but that do not occur in the object being deserialized are discarded. Classes that occur in the object but are not present in the stream have their fields initialized to default values by the default serialization mechanism.

   Externalizable objects: first the no-argument constructor is invoked and then the readExternal() method is called to restore the object's contents.

10. If enabled by a previous call to enableResolveObject(), the resolveObject() method is called. This allows the subclasses to replace the object being returned if it is necessary to do so. The result of this method is returned as the return value of readObject().

Again, the complexity in the deserialization routine can be blamed on its generality. Users could improve the performance by writing their own deserialization code.

4.4 Limitations

Even though the reasons for forcing the programmer to declare which classes should be serializable are valid, they introduce a problem. Specifically, for classes that will be serialized (e.g. those whose objects will be remote objects) special attention must be paid to their inheritance chain. If a class in the inheritance chain does not implement Serializable, its subtype should take responsibility for the serialization of its public and protected fields. As discussed in the previous sections, this is only possible if the supertype implements a constructor with no arguments. Moreover, if classes are created extending other classes from a third party library that was developed without thinking about serialization, problems may arise. This constitutes a serious problem for the development of parallelizing tools since it cannot be assumed that such libraries will not be utilized in the user's code.

Since our parallelizing tool will build distributed applications from the user's sequential code, it makes sense to make every class Serializable by default. In this way, objects can be moved among the different nodes using the default serialization mechanisms. Furthermore, performance can be fine-tuned by overloading the writeObject() method where needed (e.g. to send only a part of an array).

A way of accomplishing this change of defaults is to modify the serialization routines in the distribution source files. In particular, the files ObjectOutputStream.java and ObjectStreamClass.java could be edited to eliminate the checking that is performed to ensure that the class being serialized implements Serializable. These modifications are simple to make and are very localized but, unfortunately, things can still go wrong. In particular, if we are extending a class that implements Serializable and provides an empty writeObject() method, the object will not be properly serialized. A simple way of avoiding this is to modify the source code to only call writeObject() for the classes that we want (i.e. those that we want to fine-tune for performance) and not for the others.

Another problem that needs to be addressed is the serialization of transient and static fields. While transient fields could be left out of the serialization format (after all that is the purpose of the transient keyword), static fields are regularly used in sequential applications and extra support should be added. Since there is only one copy of a static field that is shared by all the instances of a class (which can be potentially distributed among several nodes), some runtime support will be needed to manage it. A possible solution is to have one remote object responsible for managing the static fields of a class, and to place a reference to this object in the stream instead of the field's value. This issue is further discussed in Chapter 9.

The past two sections described in detail the serialization process. From there it can be suspected that the performance of RMI will suffer because of the complexity of the serialization process (see Chapter 5). In particular, looking at the algorithm for serialization described in Section 4.2, it can be seen that each one of the ten points corresponds to an "if" statement that handles a particular case. Many of these cases cannot be avoided, but some of them could be eliminated at the expense of losing some functionality. For example, if object replacement is discarded, step 8 and part of step 3 can be eliminated. On the same track, if versioning of classes is not supported (which is not needed for parallel systems), then the computing of the serialVersionUID could be avoided together with the code that handles the evolution of types when deserializing an object. Other optimizations are possible, including custom serialization routines, and the use of *shallow* copies where possible to reduce the amount of data that needs to be exchanged.

As shown in this chapter, the problem of tagging classes as Serializable is purely a Java issue. On the other hand, the issue of how to manage static fields will arise in any object-oriented language that supports class variables such as Smalltalk [GR83] and C++ [Str91]. However, in Smalltalk this problem is easier to solve because an object can only access the fields of another object by using methods. Furthermore, this issue is also present in procedural languages like C when dealing with global data structures.

Chapter 5

Performance Evaluation

To decide how to distribute a parallel application among the available processing nodes, it is imperative to consider several factors such as processor availability and workload, security policies in place, and, in particular, the costs involved in transforming a local object to a remote one. It is important to get an exact appraisal of these costs to make intelligent decisions about the partitioning of the application and the distribution of its objects. Clearly, if creating a remote object takes longer than the time that the object is expected to run, it is wiser to create it locally. Also, after an object is remotely created, all the remote methods invoked on that object will incur extra communication overhead. This means that if some methods of a particular object are called frequently, then this object is likely not a good candidate to be made remote. Moreover, two objects which frequently talk to each other are likely to be good candidates for clustering.

Several tests were designed to obtain some reference values for the main components of the RMI system's overhead. The cost of invoking a method on a remote object can be decomposed as:

RMI pure overhead: The time that the RMI system uses to do administrative tasks (e.g. setting up connections, opening sockets, etc.).

Serialization time: The time that the RMI system uses to pack and unpack the objects sent in a call (i.e. parameters and return values).

Network delay: The time that it takes to transfer the data between two nodes (i.e. parameters, control information, etc.) without including the RMI pure overhead

```
public class TestRMIImpl extends UnicastRemoteObject implements TestRMI {

    public TestRMIImpl() throws RemoteException {
      super();
    }
    // Simple routines for testing passing
    // parameters of various types to remote objects
    public void nop() throws RemoteException {
    }
    public void nop(int x) throws RemoteException {
    }
    ...
    public void nop(Tree x) throws RemoteException {
    }
}
```

Figure 5.1: Remote Methods

and serialization time. This number depends on the speed of the underlying network and on the size of the messages being sent.

In this chapter we attempt to quantify the costs of these overheads.

5.1 Experiment Design

To get an idea about the performance of the RMI system, several test were run using two IBM RS/6000 355s with 64 Mb of RAM each, running AIX 4.1.4 connected through a 10Mbit Ethernet network. All the tests involved timing different method invocations on a remote object. Figure 5.1 shows the code for some of the remote methods used. A different empty method was provided for each type of argument that was tested (e.g. int, double, etc.). To get more accurate results the tests were run on idle machines and each measure was the average of 1000 method invocations that followed the form of the code in Figure 5.2.

To better appreciate the overhead produced by the RMI system, the same tests were run using a socket implementation (see Figure 5.3). In this version, the object serialization routines provided by Java were used to implement the remote call. Each method was assigned a number, and the call was made by writing that number

```
long tb, ta;
long et;
int it = 1000;
// Call a remote method 1000 times and time it
tb = System.currentTimeMillis();
for (int i = 1; i ≤ it; i++) {
  RemoteObject.SomeMethod(Parameters);
}
ta = System.currentTimeMillis();
et = ta - tb;

System.out.println("Avg. round trip for SomeMethod(Parameters): "+et/it+"ms");
```

Figure 5.2: Elapsed Time Algorithm

```
Socket s;
ObjectOutputStream sout;
ObjectInputStream sin;
long tb, ta, et;
long it = 1000;

// Create socket and output stream
s = new Socket(InetAddress.getLocalHost(), 6010);
s.setTcpNoDelay(true);
sout = new ObjectOutputStream(s.getOutputStream());
sin = new ObjectInputStream(s.getInputStream());

// Time 1000 socket calls
tb = System.currentTimeMillis();
for (int i = 1; i ≤ it; i++) {
    sout.writeInt(OPERATION_CODE);          // first write the operation code
    sout.writeObject(PARAMETER)                // then write the parameters
    sout.flush();
    sin.readInt();                    // wait until the method has been excecuted
}
ta = System.currentTimeMillis();
et = ta - tb;

System.out.println("Avg. round trip for SomeMethod(Parameters):"+et/it+("ms");
...
s.close();
```

Figure 5.3: Sockets Implementation

	one node		two nodes	
parameter	sockets (ms)	RMI (ms)	sockets (ms)	RMI (ms)
no parameter	5	14	5	17
int	6	16	6	16
double	7	15	7	17
Integer	7	25	7	28
Double	7	25	7	29
int[]	8	62	7	73
double[]	7	87	7	88
Integer[]	7	251	7	230
Double[]	7	281	7	235
Tree	8	387	8	362

Table 5.1: Experimental Results

to an ObjectOutputStream (that was previously bound to a socket) followed by the appropriate parameter. Then the result value of the method was read from an ObjectInputStream bound to the socket (since all the methods had a void return value, an int was used as an acknowledgment).

5.2 Experimental Results

The results of the experiments are summarized in Table 5.1. The minimum cost of an RMI call occurs when an empty method is executed on a single node. The experiments show that the minimum performance overhead incurred by the RMI system falls in the neighborhood of 14ms. This overhead clearly implies that any computation that is expected to run for less than 14ms is not a good candidate to be performed remotely using RMI (although it can still be moved for other reasons such as fault tolerance). Using an empty RMI call permitted us to isolate the pure overhead from the serialization time, as there were no parameters or return value to serialize. Likewise, the pure overhead was isolated from the network delay by running the test on one machine alone (i.e. two Java virtual machines in one physical machine).

The implementation that used just sockets and object serialization, on the other hand, took only 5ms to perform the no-argument call. This suggests a significant inefficiency in the RMI system. Unfortunately, the design of the reference and transport layers of the RMI system is not documented (only the interfaces are documented). Furthermore, the source files of the classes that implement these layers are not provided with the Java distribution. Thus, it is almost impossible to fully explain where the time is spent inside the RMI layers. However, we used a decompiler to take a look at the main classes that comprise these layers to get a better understanding of the RMI system. The exploration of these classes revealed a fairly complex design. Moreover, this design cannot be compared to the sockets implementation that we made. Besides creating sockets, the RMI implementation creates several threads and performs several other tasks, such as protocol negotiations and garbage collection. For example, one of the threads created is a Pinger thread whose function consists of sending periodic messages to the other side of a communication channel to test the liveness of the connection. Another interesting thing we noticed is the presence of debugging code. In particular all the code is populated with "if" statements that test for different properties and output the appropriate information to log-files. Clearly, these "if" statements contribute to the poor performance of the RMI system.

Since the socket implementation and the RMI implementation both used the same pack and unpack code (i.e. the object serialization feature of Java), the difference in our test program times can be attributed solely to the RMI overhead. As discussed, this overhead is attributed to debugging code, thread management, thread context switch, etc. However, the user of RMI benefits from the RMI's ease of use. Invoking a remote method using RMI is much easier than programming the equivalent socket version.

To test the performance of the serialization routines of RMI, empty remote methods, as shown in Figure 5.1, were used. Different runs were made changing the parameter type to see how this affected the performance. Two primitive data types of different size were tried: int (32 bits) and double (64 bits). As shown in the table, the time that it takes to serialize and send a primitive data type across the network is insignificant compared with the RMI pure overhead.

Other types were also used as parameters. In particular, objects of type Integer and Double were used to compare the times with the ones obtained for the equivalent primitive data types. The numbers in Table 5.1 show that it is much more expensive to send an Integer (Double) object than to transmit the corresponding int (double) primitive data type. This is because the serialization of objects is more complex than the serialization of primitive data types. For objects, class information is placed in the stream and extra dispatches are performed to pack the object's contents (see Section 4.2). However, the socket version seems less affected by this than the RMI version.

More complex structures were used to see how the RMI performance was affected. Four different arrays of 100 elements were tried: an array of int numbers, an array of double numbers, an array of Integer objects, and an array of Double objects. Also, a binary tree made of 100 objects, each containing an integer number, was used as a parameter. These experiments confirmed the fact that dealing with objects is more expensive than dealing with primitive data types. Also, serializing the tree took longer than serializing an equivalent array. This is explained by the fact that each node of the tree contained two object references besides the int value. For each of these references a handle was written. On the other hand, when serializing the array, its size is written once followed by all the elements (see Section 4.2). While the performance of the socket implementation remained almost constant for the different types, the performance of RMI deteriorated.

The tests were also run on two nodes with similar results. This implies that the network delay is not a significant factor in the total overhead. In other words, for the types used as parameters, no performance improvement can be achieved by using a faster network. The numbers for two nodes also show how the RMI system benefited from overlapping computation and communication. When sending an array of integers for example, while one node is serializing to the stream, the other node is deserializing from it. This explains the fact that some method invocations were faster through the network than locally.

5.3 Conclusions

This chapter presented a performance evaluation of the RMI system. The experiments revealed an important performance overhead in the RMI system. This overhead cannot be attributed to the serialization routines but is directly caused by the RMI reference and transport layers internal workings. Debugging code present in those layers contributes to the performance degradation.

The default serialization routines (see Section 4.2) could also be made faster by eliminating some features like class versioning and object replacement that hurt the performance and are not needed for parallel programming (of course they provide functionality that is needed for other types of applications). The experiments also show that the serialization of objects is more expensive than the serialization of equivalent primitive data types. This happens because the default serialization routines, when serializing an object, write information about the object's class, the fields of the object's ancestors, etc., and perform some extra manipulations (e.g. object replacement, string conversions) that are not done when dealing with primitive data types.

Also, special care must be taken to serialize exactly what is needed. For example, the default serialization routines take a *deep copy* approach and whole object graphs are sent. Clearly, sending a 1000 position array to work with only 10 positions at the remote site is a very inefficient solution; as opposed to the *shallow copy* approach were only what is needed is actually sent.

As shown by the experimental results, the implementation of the RMI system is the bottleneck of the object distribution. This fact is confirmed by some systems like Voyager [VO97] that also use serialization but are much faster than RMI (see [VR97] for a performance comparison between Voyager and RMI). By using RMI the programmer trades performance for ease of use.

Faster networks will not solve the problem. JIT compilers can be used to increase the performance of all the RMI code. Also, some code clean-up would be beneficial (at least the debugging code should be eliminated). However, the gains would be marginal because the complexity of the code will still remain the same.

Chapter 6

Inheritance

Inheritance is a very important feature of object-oriented languages that enables the sharing of code between different class implementations. When used properly it is a powerful tool for software development.

Java supports tree single inheritance where each class has exactly one parent and all classes have a common ancestor (i.e. the class Object). From the simplicity point of view, single inheritance provides an unambiguous model that is easy to deal with (as seen in languages such as Smalltalk [GR83]). However, sometimes this model lacks expressiveness when compared to those languages that support multiple inheritance (e.g. C++ [Str91]). The Java approach for dealing with this issue is through *interfaces*, which regain some of the lost expressiveness without sacrificing the simplicity. However, using interfaces sometimes forces the programmer to unnecessarily duplicate code.

6.1 Inheritance and RMI

The simplest way of implementing a remote object with RMI is by extending the class UnicastRemoteObject. However, since Java provides only single inheritance this is not always possible. Moreover, if we plan to automatically translate local classes into remote ones, we cannot rely on this feature because the user classes often extend some other classes. For example, suppose that the user has a class AccessPermits that is implemented as an extension to the class java.util.BitSet which provides a growable vector of bits. If we have to make objects of type AccessPermits remote, we cannot extend UnicastRemoteObject because the only available inheritance path is already consumed by java.util.BitSet.

Another option available is to use the static method exportObject() provided by class UnicastRemoteObject to make the remote object available for receiving incoming calls. In our previous example we would create an instance AccessPermits and then we would export this object using a UnicastRemoteObject.exportObject() call as in the following code fragment:

```
...
AccessPermits permits = new AccessPermits();
...
UnicastRemoteObject.exportObject(permits);
```

Using this approach, however, requires some extra coding to implement the semantics of Object that are different for remote objects, that is, the hashCode(), equals(), and toString() methods. In the first approach these methods were inherited from RemoteObject but since inheritance cannot be used for the second approach they have to be re-implemented. [1] Even though the code for these methods can be copied from the RemoteObject.java source file that comes in the Java distribution, the code will be repeated for each user class that is made remote. This, besides being inconvenient, is clearly less efficient in terms of space.

6.2 Inherited Methods

In the RMI system, only the methods declared in the remote interface of an object can be remotely called. Thus, inherited methods cannot be called directly from a remote site. Consider the inheritance graph shown in Figure 6.1. A Sensor class provides the common functionality (e.g. Activate() and Deactivate()) to different types of sensors (e.g. TempSensor, NoiseSensor, etc.).

Now, let's say that we make instances of TempSensor remote. According to the RMI specification, all the methods that will be called remotely should be put in the remote interface of the class. So if the TempSensor class had, for example, a method for reading the temperature (i.e. readTemp()), the remote interface should look like the code in Figure 6.2.

[1] In fact they only have to be implemented if they are used.

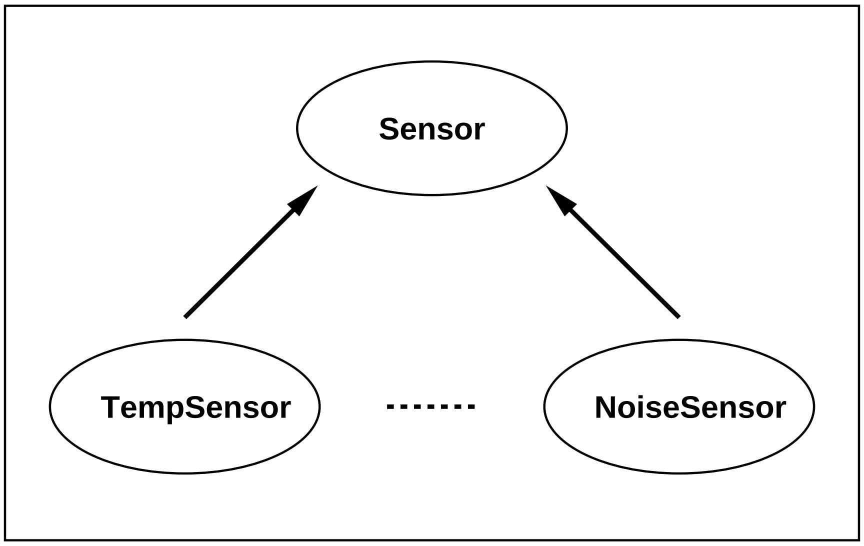

Figure 6.1: Sensor Inheritance Hierarchy

```
public interface TempSensorRI extends Remote {
    Temp readTemp() throws RemoteException;
}
```

Figure 6.2: Temperature Sensor Remote Interface

Figure 6.3 shows a code fragment that uses a local TempSensor. Basically the code instantiates a TempSensor, activates the sensor, reads the temperature and finally deactivates the sensor.

When the TempSensor is made a remote object, the code shown in Figure 6.3 changes to something similar to the code fragment shown in Figure 6.4. Note that in the remote object version "ts" is a TempSensorRI rather than a TempSensor. Since TempSensorRI does not contain the methods Activate() and Deactivate(), an error will be generated when we try to compile this code. For this program to work Activate()

```
...
TempSensor ts = new TempSensor();
ts.Activate();
System.out.println("Temperature: " + ts.readTemp());
ts.Deactivate();
...
```

Figure 6.3: Local Use of TempSensor

```
...
Registry registry = LocateRegistry.getRegistry();
TempSensorRI ts = new (TempSensorRI) registry.lookup("TEMP_SENSOR");
ts.Activate();
System.out.println("Temperature: " + ts.readTemp());
ts.Deactivate();
...
```

Figure 6.4: Use of TempSensor as a Remote Object

and Deactivate() should be added to the remote interface of the temperature sensor (i.e. to the TempSensorRI interface shown in Figure 6.2).

This simple example demonstrates that when converting a regular object into a distributed one, all the inherited methods should also be put in the remote interface. Automatically doing this, however, would require compiler support to find all the methods that must be added to the remote interface. The inheritance hierarchy should be traversed from the class we are working on up to the Object class and the inherited methods should be added as we go along. However, since Java's method dispatching is done according to the runtime type of the receiver, the compiler will not always be able to determine if a particular inherited method will be used or not. This means that the interface will likely include methods that will never be called. To illustrate this suppose we have a class **B** that extends another class **A**, and that **A** implements a method **alpha()**. Suppose also that the compiler is parsing the following code:

```
B b = new B();
A a = b; // this is valid because A is a supertype of B
...
a.alpha();
```

The variable "a" has a compile type of **A** but its runtime type is **B**. The compiler usually cannot determine this because instead of a simple assignment, "a" can be the result of a more complex operation. By just looking at the code the compiler will not be able to tell if alpha() will be called or not for object "b". But, if "b" has to be made remote, alpha() should be added to the remote interface just in case it gets called as in the sample code.

```
public class Sensor {
    protected long serial_no;
    ...
}
```

Figure 6.5: Sensor Class

```
...
NoiseSensor ns = new NoiseSensor();
if (ns.serial_no == 12345)
    do_something_special();
else
    ...
```

Figure 6.6: Access to a Protected Attribute

6.3 Inherited Data

Inheritance should also be considered from the point of view of the inherited data. Suppose that we have the inheritance hierarchy shown in Figure 6.1 and that the Sensor class has an attribute to keep the serial number of a particular sensor (see Figure 6.5). The attribute serial_no is declared protected so it can be accessed from the Sensor class, from all its subclasses (i.e. TempSensor, NoiseSensor, etc.) and from all the classes that belong to the same package. [2] So we could potentially have a class in the same package which has some code like the sample in Figure 6.6. When NoiseSensor is made remote however, the attribute cannot be directly accessed anymore. Code like that in Figure 6.7 will produce a compiler error because the attribute cannot be part of the remote interface.

Since the code in Figure 6.7 uses the remote interface as a type and since variables cannot be declared in an interface (only constants and methods), the inherited value is not known by ns. This happens because RMI forces you to instantiate the interface rather than the remote class so only those things defined in the remote interface (NoiseSensorRI in our example) are visible from the clients. A simple workaround of

[2] This differs from C++ where protected fields are only accessible from the class where they appear and from their subclasses. To get the same functionality in Java the *private protected* modifier should be used instead.

```
...
NoiseSensorRI ns = (NoiseSensorRI) registry.lookup("NOISE_SENSOR");
if (ns.serial_no == 12345)                    // <— COMPILER ERROR!!!
    do_something_special();
else
    ...
```

Figure 6.7: Remote Access to a Protected Attribute

this problem involves changing all the data accesses to remote method invocations. To perform these changes automatically, compiler technology will be required to detect all the data accesses and turn them into the appropriate set() and get() method calls. Specifically, every place in the code where a field is used on the left-hand side of an assignment should be replaced by a call to a set() method; any other use of the field should be replaced with a get() method. Even though this problem is shown in the context of inheritance, it is not the inheritance *per se* that causes it. This problem is caused by the lack of encapsulation in the language. If all the data were private to the objects, and accessed only through methods, it would be enough to convert those methods to remote.

6.4 Conclusions

As shown in the previous sections, inheritance raises many issues when combined with remote objects. The first issue is how to utilize RMI in an application that uses inheritance. As shown, the single inheritance model supported by Java is inconvenient because it forces the programmer to choose between unnecessarily duplicating code or to derive his/her classes from the RMI base class. Moreover, when the translation from local to remote objects is automatically made, the compiler is presented with the same decision. Both, the programmer and the compiler, should extend the RMI base class whenever possible to avoid duplicating code. This problem can be directly mapped to any language that is restricted to single inheritance. In those languages the programmer is always faced with the decision of which inheritance path is better to use.

On the other hand, the problem of adding the inherited methods to the remote interface is specifically RMI related (HORB [Sat96] and Voyager [VO97] do not have this problem). The stub generator should directly add the inherited methods to the remote interface. However, doing so could considerably increase the size and complexity of the stubs and skeletons generated. The solution adopted by the developers of RMI is that the programmer should not be penalized (in terms of space and/or speed) for something that he/she does not use (i.e. inherited methods that will never be called remotely). Unfortunately, when the translation from local to remote is automated, this optimization becomes a source of conflict.

Finally, the problem that arises with the inherited data is a language issue. If the semantics of the language were such that the encapsulation of data could not be violated with direct access to the object's fields, the problem would vanish. In that case, all the data accesses would be made through methods that would be transformed into remote methods when passing from local to remote objects. The performance hit suffered when doing a local method call instead of a direct memory access could be lessened by using compiler optimization techniques such as *code inlining*. Note, however, that since Java supports dynamic method dispatching, more complex code inlining techniques such as polymorphic inline caching [HCU91] will be required.

Chapter 7

Futures

Usually, in a concurrent system, achieving maximum performance involves executing as many concurrent tasks as possible. To increase concurrency, it is often desirable to synchronize only when it is absolutely necessary; that is, the blocking required to synchronize a task should be delayed as much as possible. Consider the following code:

```
...
x = x_ExpensiveComputation();
y = y_ExpensiveComputation();
...
System.out.println("x = " + x + " y = " + y);
```

Suppose that the value of the variable x is not needed to compute the value of variable y. If the sequential semantics are respected, y_ExpensiveComputation() cannot start until x_ExpensiveComputation() has finished. However, it would be highly beneficial if both values could be calculated in parallel. The only restriction would be that *both* computations should finish *before* the values are printed.

To run both methods concurrently, two threads have to be created and a synchronization point has to be set up to wait for both computations to finish before executing the print statement. All this can be elegantly expressed using a construct called a *future* [BH77] [Hal85]. A future is a commitment to use a particular value at some later time. Whenever a future is created, a new thread is spawned to run that part of the computation and, eventually, fill in the value for the future. At a later time, when the original thread tries to access this future, it will block until the future is resolved (i.e. the value becomes available). Thus, in our example, it would be enough to declare x and y as *futures* to obtain a correct concurrent execution. The following code shows our example using futures:

```
Future x, y;
...
x = CreateFuture(remote_node(), x_ExpensiveComputation());
// continue without blocking
y = CreateFuture(remote_node(), y_ExpensiveComputation());
// continue without blocking
...
WaitForFuture(x); // we block only when it is needed
WaitForFuture(y);
System.out.println("x = " + x + " y = " + y);
```

Futures are a popular synchronization mechanism used in variety of parallel programming systems such as Enterprise [SS+93] and Mentat [Gri93]. They were first introduced by Baker and Hewitt [BH77] and later on became popular in Multilisp [Hal85].

7.1 Futures in Java

The RMI system does not provide an asynchronous method calling facility (i.e. all remote method invocations are synchronous). We evaluated three different approaches to implement futures in Java so more parallelism could be obtained by using asynchronous calls. We started from the most intuitive solution and improved from there.

As we saw, creating a future implies spawning a new thread for executing the computation. This thread can be created on the caller or on the callee side, and there are many implications that affect this decision. Three approaches investigated in this thesis explore these issues. In all of them a Future object is created. This Future object is a remote object which provides a value holder for the future and provides methods to access it (Figure 7.1). Each Future object also contains a flag

```
package Futures;
import java.rmi.*;

public interface FutureRI extends Remote {
        void setValue(Object o) throws RemoteException;
        Object getValue() throws RemoteException;
}
```

Figure 7.1: Future's Remote Interface

that is used to record the state in which the future is at any particular moment (e.g. unresolved, resolved, error). Being a remote object, the Future object can be passed around among different virtual machines and its value can be retrieved from where it is needed. This constitutes a step forward from some designs where the futures are not able to move (e.g. Enterprise [SS+93]).

7.1.1 Server Side Futures

In the first approach, called *Server Side Futures*, the Future object and the extra thread are created on the server side (see Figure 7.2). Whenever a client executes a remote method (e.g. compute()) on the server, the server spawns a new thread to execute the method and immediately returns a Future object to the client. [1] At this point the client can continue executing concurrently with the computation that is running on the server side without blocking. The server is responsible for updating the value in the Future object whenever it becomes ready by using the method setValue(), which is part of the Future's remote interface (see Figure 7.1). The method setValue(), besides storing the value of the future, wakes up all the threads, if any, that are blocked waiting for the future. If the client tries to access the value in the future, using the getValue() method, before the future has been resolved (i.e. before the server updates it), the client will block until the value becomes available. This blocking is done by using the thread synchronization primitives of Java. Inside getValue(), wait() is used, if necessary, to suspend the executing thread. This, in turn, will stop the method getValue() from returning until the thread is awakened by a notifyAll() call

[1] Note that since the future is a remote object, it is passed by reference and, thus, only a handle is returned to the client.

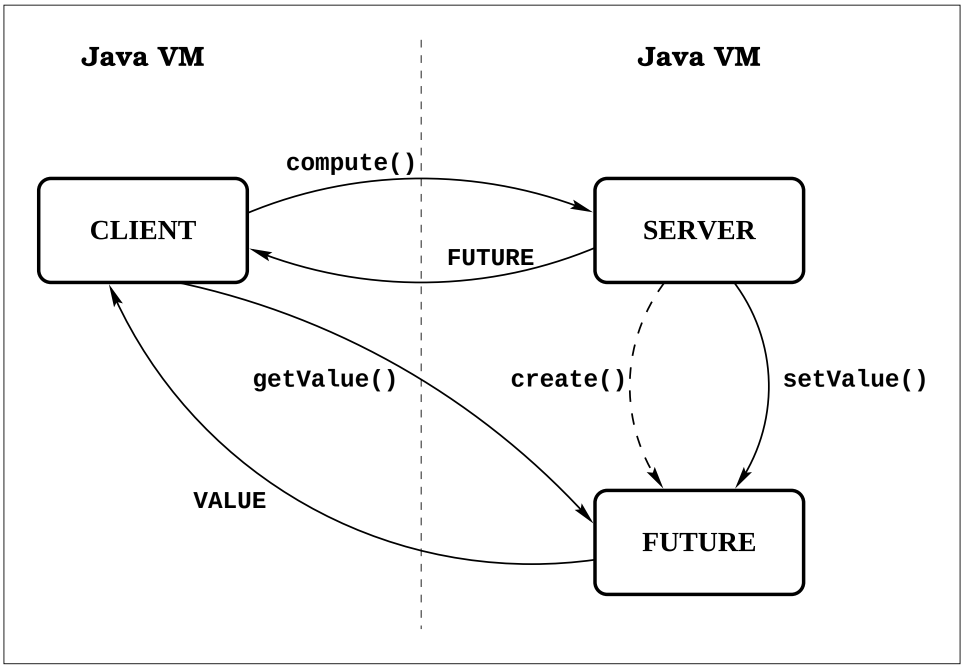

Figure 7.2: Server Side Futures

in setValue(). If for some reason the computation is aborted on the server side, the client trying to get the value from the future will get an exception instead. See Figure 7.3 for sample code.

7.1.2 Client Side Futures

The second approach, called *Client Side Futures* (see Figure 7.4), is an optimization over the *Server Side Futures* approach. Even though both are semantically equivalent, the *Client Side Futures* approach reduces the inter-virtual machine messages necessary to create the Future object and start the computation. In this implementation, the future is created on the client side and then passed to the server as an extra parameter of the remote method invocation. Note that even though the Future object is created on the client side, the thread needed for executing the computation is still created on the server side. Since the future itself is a remote object, the extra overhead of sending another parameter is minimal because remote objects are passed by reference (i.e. only a handle is sent). The semantics for accessing the Future object remain the same. On the server side the code is similar to that of the previous approach with

```
//—[ Client Code ]—
try {
    Registry registry = LocateRegistry.getRegistry();
    serverRI server = (serverRI) registry.loookup("SERVER");

    FutureRI f = server.compute();
    // Client continues executing...

    // Client will block when accessing the value.
    Integer i = (Integer) f.getValue();

} catch(Throwable t) {
    // Error handling code...
}


//—[ Server Code ]—
private int func;                                    // A different ID for each method.
private Remote f;

public synchronized FutureRI compute() throws RemoteException {
  func = 1;                                            // set method ID for execution.
  f = (FutureRI) new Future();
  new Thread(this).start();
  return(f);
}
public void run() {
  Thread.currentThread().yield();
  switch(func) {                       // identify which method should be executed.
    case 1: {                                                    // static dispatch.
      int i = real_compute();                    // the original code for compute()
      try {
          f.setValue(new Integer(i));
      }catch(Throwable e) {
          // Error handling code...
      }
    }
    case 2:
      // a case for each method remote method that has to be
      // executed in a separate thread.
  }
}
```

Figure 7.3: Code for Server Side Futures

the exception that instead of instantiating a Future it uses the instance provided by the client. On the client side the code looks like:

```
...
FutureRI f = new Future();
srv.compute(f); // srv is a reference to the server object
...             // which resides on a different virtual machine.
```

Both approaches require changes to the server side code to spawn a new thread for executing the remote method. This is necessary because the method is required to return immediately without blocking. Furthermore, since Java threads only execute the code placed in the run() method, it is necessary to make a static dispatch inside the server for each remote method we plan to execute in a separate thread (see the switch in the run() method of Figure 7.3). This makes the required changes to the user code more difficult. Also, the signature of the remote method should be changed to return a Future in the first approach and to include an extra parameter in the second (besides changing the return value to void).

7.1.3 Client Side Futures / Future Computes

The third approach, called *Client Side Futures / Future Computes* (see Figure 7.5), is an attempt to alleviate all the problems stated in the previous section (i.e. to minimize the required changes to the user's code and to eliminate the static dispatch inside run()). In this case, the future is created on the client side and it is responsible for executing the remote method. Now, the server code remains unchanged and the future is a threaded object that will block on behalf of the client. Note that this time the thread used for running the computation is created on the client side. Another way of achieving this would have been by making the client a threaded object. However, Java imposes the restriction that run() is the only method that a thread can execute, forcing us to modify the client code in the same way as we did with the server code in the previous two approaches. That is, a static dispatch should be coded inside the client's run() method. Creating the thread on the future itself eliminates all these problems. When the future is created it receives a reference to

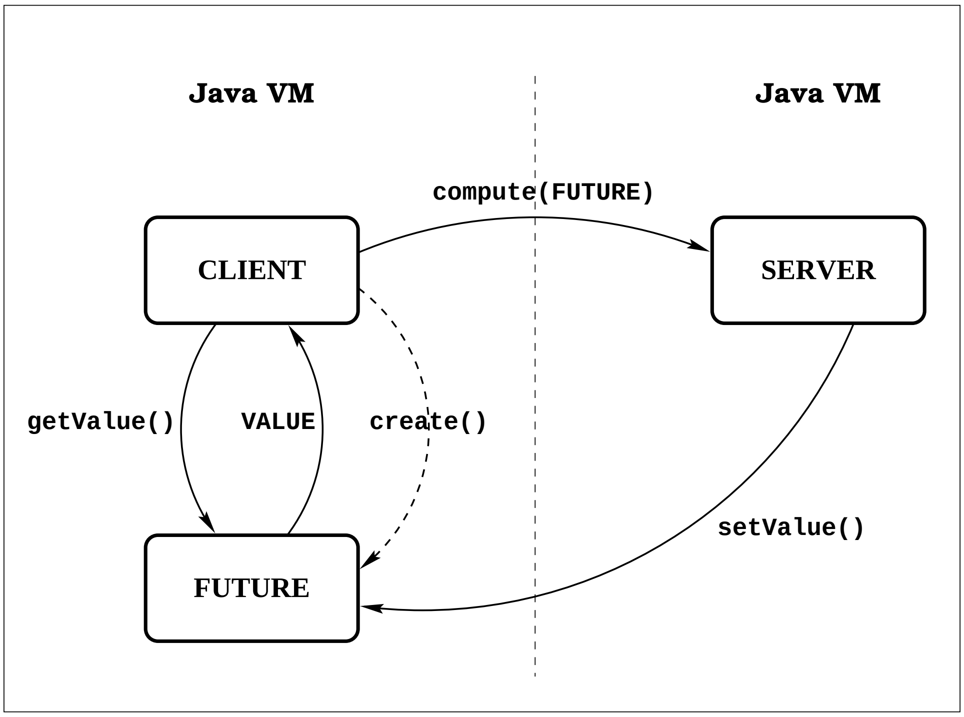

Figure 7.4: Client Side Futures

the server and a method signature to execute on that server. The future will use the Reflection API to make the call (see Figure 7.6). The Reflection API provides the functionality necessary to gather information about fields, methods and constructors of loaded classes, and to operate on them. In particular, in the code shown, it is used to execute a specific method on the server with the specified parameters.

This approach has the advantage of keeping the server code untouched. However, executing a method through the Reflection API is more expensive because it involves a table lookup for finding a method in the server's class that matches the specified signature, an object instantiation to create a `Method` object that "reflects" the specified method, and a dispatch to execute this method on the desired object (see the `run()` method in Figure 7.6).

A possible workaround of this performance problem consists of subclassing the `Future` class. The `Future` class should be changed to be an abstract class with an empty method, called `exec()`, that will be called inside `run()`. Each time a future is needed to asynchronously execute a method, an appropriate subclass of `Future` should be used. This subclass should provide the implementation of `exec()` which will execute

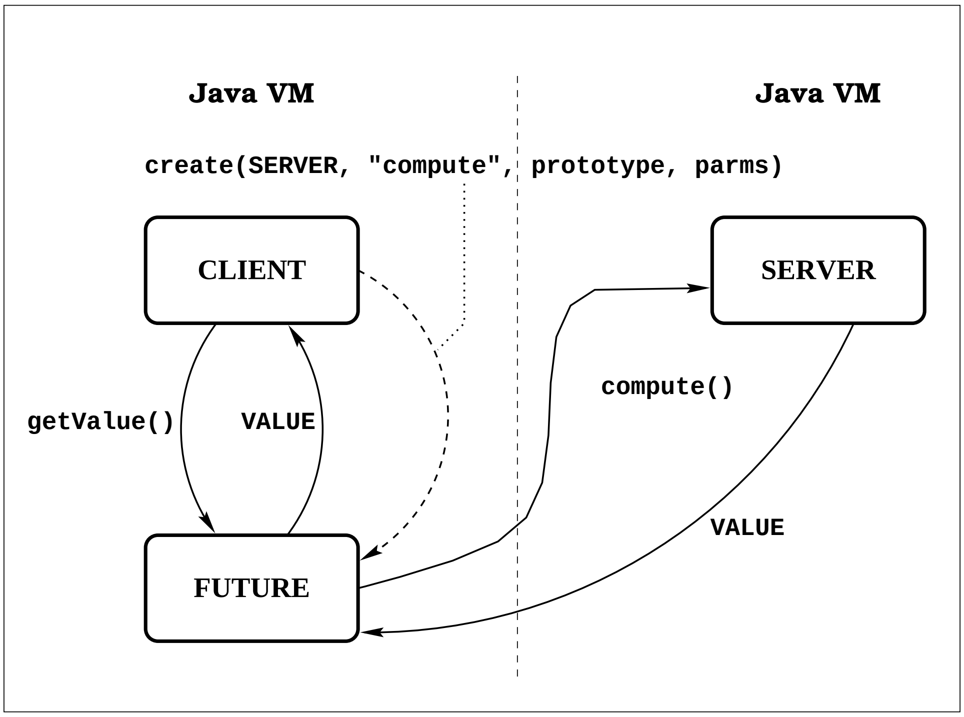

Figure 7.5: Client Side Futures / Future Computes

the desired method. The rest of the functionality will be inherited from the Future class. Even though this subclassing can be inconvenient for a programmer, it can be easily performed by the parallelizing compiler.

7.2 Limitations

Regardless of the approach chosen for implementing futures in Java, some common problems also need to be addressed. The first problem is how to stop a computation once it is running. Suppose that an application searches a binary tree looking for a specific value. One possible implementation might have two futures created: one for a computation that performs the search on the right branch, and the other for searching the left branch (speculative computing). Clearly, if one of the computations returns having found the value (i.e. the future is resolved), then the other future can be discarded. A simple way of dealing with this consists of doing nothing; that is, taking the first result and ignoring the second. Even though this solution is simple to implement, it has the undesirable effect of wasting CPU cycles to compute a value that will never be used. An alternative to this involves providing a method to stop

```
public class Future extends UnicastRemoteObject implements FutureRI, Runnable {
    // attribute declarations...
    public Future(Remote srv, String name, Class _prototype[], Object _parms[])
            throws RemoteException {
        super();
        // Since run() doesn't have parameters the data is kept on attributes.
        value = null;
        state = Future.NOT_READY;
        server = srv;
        method = name;
        prototype = _prototype;
        parms = _parms;
        new Thread(this).start();             // Creates and starts a new thread.
        return;
    }
    public Object getValue() throws RemoteException {
        while (state == Future.NOT_READY) {
            try {
                wait();                              // block the client thread.
            } catch (InterruptedException e) {}
        }
        return(value);
    }
    public void run() {
        Thread.currentThread().yield();
        Object o = null;
        Method m;
        try {
            // Get a method object that matches the prototype
            // on the class of the server.
            m = server.getClass().getMethod(method, prototype);
            // Invoke the method on the server with the saved parameters.
            o = m.invoke(server, parms);
        }catch (Throwable t) {
            System.out.println(t.getMessage());
            t.printStackTrace();
        }
        value = o;                              // Set the value of the future.
        state = Future.READY;                     // Mark the future as ready.
        notifyAll();                           // Wake up all waiting threads.
    }
}
```

Figure 7.6: Future's Code for Client Side Futures / Future Computes

the desired computation. This method should be implemented in the server or in the future according to which approach has been taken for implementing the futures. If the thread that is executing the computation has been created by the server (i.e. *Server Side Futures* or *Client Side Futures*), the method for stopping it should be added to the server's class. If the thread has been created by the Future object (i.e. *Client Side Futures / Future Computes*), the Future class should provide this method. Regardless of its location, the implementation of this method should stop the thread that is running the computation by sending it the message stop(). [2]

Another problem that requires attention involves exceptions. Exceptions are a common feature in many modern languages such as C++ [Str91]. Suppose we have the following sequential code:

```
try {
  Integer i = server.compute();
  Use(i);
} catch(SomeException e) {
  Do something with e...
}
```

If the compute() function throws an exception, control skips to the catch clause, and the function Use(i) never gets called. Now if "i" is made a Future and passed to Use(i), when the exception is thrown on the server, it is too late to recover. That is, by the time that the exception is thrown, Use(i) could have made some changes and/or performed certain actions that could not be reversed. A partial solution to this problem consists of using two futures to represent the value calculated by compute(). The first future corresponds to the value "i"; the second represents the *return code* for the function. The idea is to block in the future representing the return code instead of the future representing the value. The future representing the return code will be resolved as soon as the server passes the place where it can throw the exception. The code in the client will look like this:

[2] The message stop() is understood by every Java thread and forces it to stop executing.

```
FutureRI rc;
FutureRI i  = server.compute(rc);
// Block until the server signals that no exceptions can happen.
Exception e = (Exception) rc.getValue();
if (e) // if an exception occurred...
   Do something with e...
else
   Use(i);
```

This technique of dividing the server code in two parts, one that can throw exceptions and another that cannot, allows some concurrency while retaining the correct sequential semantics for the code. However, extensive compiler support will be needed for automating this task. The compiler should identify the point in the server code were no more exceptions can be thrown and resolve the future that holds the return code.

An alternative solution consists of using transactions similar to those which are common in modern databases. The state of the objects should be saved before every change and rolled back as needed (i.e. when an exception is thrown). Although semantically correct, this solution would be hard to implement and the performance would probably be unacceptable.

The simple example presented in this section demonstrates that exceptions can seriously reduce the opportunities for concurrency. Since the use of exceptions is strongly encouraged in Java, creating concurrency while preserving the original sequential semantics becomes very difficult. This posses a serious problem for the development of an automatic parallelization tool. Moreover, since exceptions are a common feature in many modern languages this problem is not limited to Java alone.

7.3 Discussion

As shown in the preceding text, it is generally better to create the Future object on the client side. This reduces the inter-Virtual-Machine messages at creation time resulting in a more efficient design from the performance point of view. However, creating the Future object on the server side may still be desired for load balancing reasons (e.g. the client machine may have too many objects, or may be running out of memory, etc.).

The complexity needed on the server side to asynchronously execute different actions can be avoided using the *Client Side Futures/ Future Computes* approach. The `Future` object is responsible for executing the remote method and the Reflection API is used as a generic dispatching mechanism. Alternatively, the performance overhead of the Reflection API can be avoided by subclassing the `Future` class.

Special attention has to be paid to exceptions in the presence of futures. As shown, if the sequential semantics have to be preserved in the face of exceptions, then the concurrency provided by asynchronous messages may become severely limited.

It is important to point out, however, that the limitations previously described are not a flaw of either Java or the RMI system. The problem with exceptions will arise in any language that supports this particular construct (e.g. C++), and the problem of how to stop an ongoing computation is intrinsic to the parallel computing paradigm. The most serious of these two limitations is the one involving exceptions. If futures are to be used in languages that support exceptions, more research will be needed to find a better way of integrating both constructs (i.e. exceptions and futures).

Chapter 8

Collaborators

An object-oriented system can be viewed as a group of objects that communicate among themselves using messages to achieve a particular goal. We can say that these objects *collaborate* with each other to solve a particular problem towards the common objective. In this scheme, any object whose methods are invoked by another object is said to be a *collaborator* of the second.

This chapter explains how the concept of collaborators relates to the automatic parallelization of sequential programs. First, a working example is presented which is used to illustrate the impact of the different distribution decisions (i.e. what are the consequences of selecting different objects to be made remote). Then, the problem of automating these decisions is discussed.

8.1 Example

Suppose we have an object that represents a temperature sensor which is an instance of the class TempSensor shown in Figure 8.1. Also suppose we have an object of type TempDisplay (see Figure 8.2) that displays a temperature on a console screen every 30

```
public class TempSensor extends Sensor {
    public Temp readTemp() {
        Temp t = Physically_read_temperature();
        return t;
    }
}
```

Figure 8.1: Temperature Sensor

```
public class TempDisplay extends TextField implements Runnable {
    private TempSensor ts;
    public TempDisplay(TempSensor t) {
        ts = t;
    }
    public void run() {
        while (true) {                                    // do forever
            // use the temperature sensor to
            // read and display the current temperature.
            this.setText(ts.readTemp());
            try {
                wait(30000);                          // wait 30 seconds.
            } catch(InterruptedException e) {}
        }
    }
}
```

Figure 8.2: Temperature Display

seconds. Since each time an instance of TempDisplay needs to display the temperature it invokes a method in an object of class TempSensor, we can say that the objects of class TempSensor are collaborators of those of class TempDisplay.

A collaboration diagram can be drawn where the objects are represented by circles and the methods by directed arcs. Figure 8.3 shows the collaboration diagram for our temperature sensor example. Each object pointed to by an arrow is a *collaborator* of the object at the other end of that edge.

8.2 Collaborators and RMI

The purpose of this section is to show how the collaborators affect the translation of local objects into remote ones. As long as an object and its collaborators are on the same virtual machine everything works properly. Suppose now that we decide to place one of the objects on a different virtual machine. In our example let's say that we are interested in monitoring the temperature of a remote oven that is controlled by some computer. Hence, we have to make the instance of TempSensor a remote object. In doing that we have made readTemp() a remote method putting it in the remote interface of TempSensor. Moreover, the reference to a TempSensor that is held

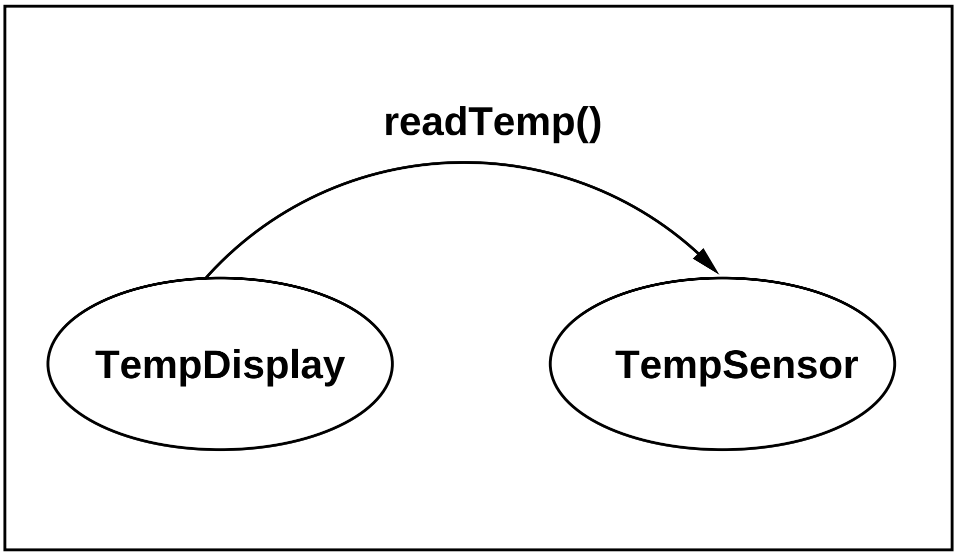

Figure 8.3: Collaboration Diagram

by the instance of TempDisplay will now be of type TempSensorRI (i.e. the remote interface). Now, both objects reside on different virtual machines but, thanks to the transparency provided by RMI, the syntax of the invocations remains the same and everything works as planned.

Now suppose that instead of choosing the instance of TempSensor as our remote object, we opted for an instance of TempDisplay. Even though this decision seems awkward and that the previous solution appears to be the "natural" choice, this new setting could be the result of a compiler (or parallelizing tool) decision. Furthermore, a parallelizing tool should be able to handle any placement strategy (i.e. any class can be chosen to make it remote). If objects of TempDisplay are remote it means that they will not be able to invoke readTemp() because the instance of TempSensor resides on a different virtual machine. Note that the constructor of TempDisplay receives a reference to an object of type TempSensor as a parameter. This reference cannot be to an object that resides on the same virtual machine because the whole point was to have both objects residing on different virtual machines. Thus, in order to make this scheme work, we need to also make the instance of TempSensor a remote object.

This clearly implies that whenever we make a sequential object remote, we also have to make all of its collaborators remote objects (more precisely, those that reside on a different virtual machine). Even though both solutions work, the first choice is better because it provides the same functionality with less resources (i.e. one remote object instead of two). Since every collaborator of a class that is made remote has to be transformed into a remote object, a *domino* effect, where most of the classes are converted, could result. It is important to reduce this effect to a minimum because the cost of invoking a remote method is much larger than the cost of executing a local method, and classes that reside on the same virtual machine should not be forced to pay an unnecessary performance penalty. In particular, a remote invocation involves the serialization and transmission of the method's parameters and return value which could be a very expensive process as discussed in Chapter 5. Note that even if the caller and the callee of a remote method both reside on the same virtual machine, this process remains the same (i.e. parameters and return values are serialized and transmitted through the RMI's layers). Suppose in our example that TempDisplay has another collaborator for getting the time of the day; let's say Timer. In the first solution, the instances of Timer will not be affected by making TempSensor remote. However, if TempDisplay is chosen instead, the instances of Timer will also be transformed into remote objects and the method for getting the time of the day will be transformed into a more expensive remote method.

8.3 The Translation Process

As shown in the previous section, all the collaborators of an object that is being converted from a local object to a remote object should also be converted. If the programmer is the one manually performing the translation, he/she will be responsible for identifying the collaborators and to make appropriate decisions about *what* goes *where*. If this decision is given to a compiler or to an automatic parallelization tool, the collaborators of a class should be automatically detected. A safe approach would be to make every object a remote object. In this way, any object can be placed anywhere and its methods can be called from any other node. Even though this approach works, performance suffers for each remote method invoked on an object that resides on the same virtual machine as the caller object.

However, if the distribution of objects is static (i.e. object migration is not supported), some optimizations can be applied. Instead of making every object a remote

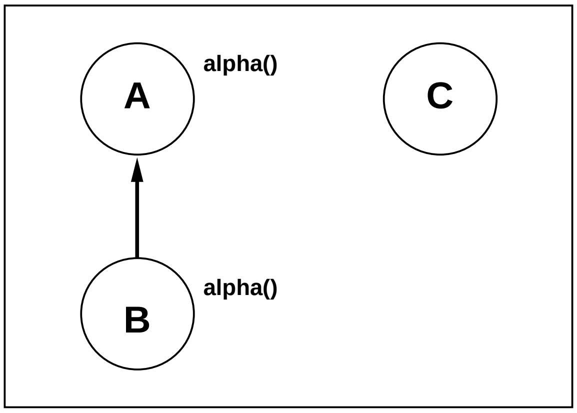

Figure 8.4: Collaborators Detection

object, only a group of selected objects together with their collaborators are made remote. In this way, the performance overhead of a remote call is avoided for unaffected local objects. To implement this, the compiler should be able to determine all the collaborators of a given class. For doing that, the source code alone will not be enough because the compiler could be fooled by the method dispatching semantics of Java and the existence of public objects. Consider the following example: three classes, A, B, and C, where class B extends class A and both implement a method alpha() (see Figure 8.4). Now suppose that the code in main() is like the following:

```
public B b = new B();
public A a = b;
```

Consider the following code in class C:

```
a.alpha();  // this invokes alpha() of class B.
```

When the compiler finds a.alpha() in class C, it can assume that A is a collaborator of C. This happens because Java dispatches according to the runtime type of the receiver, which in this case is different from the compile-time type. In this example, the compiler would have made the wrong class remote.

Clearly, the compiler alone is not enough to determine the collaborators of a class from the source code. Extra information is needed. [1] This information can be obtained from hints supplied by the user, and/or from the application structure. For

[1]Note that instead of an assignment, "a" can be the result of a more complex operation (it can even be retrieved from disk).

example, in a parallel application development framework based on patterns [GH+94], where the user develops the application by filling the code sections in the provided patterns, the structure of the patterns allows the compiler to unambiguously determine the collaborator set. For example, suppose we have a pattern that represents a pipeline with several stages. Suppose also that, according to this pattern, each stage finishes its task and then calls a method on the next stage to pass the results. From the structure of this pattern, it can be determined that each stage is a collaborator of its previous stage (except the first one of course).

If extra information is not available, a compromise solution could be adopted where the compiler makes remote the class that it detected as a possible collaborator and all of its children. In the example above, A and its children (i.e. B) will be made remote. Even though this solution produces more remote objects than really needed, it is better than making everything remote (and is just as safe).

Providing that the set of collaborators can be determined, another possible optimization concerns their methods. Even though the collaborators have to become remote objects, not all their methods need to be made remote. Only those methods that are part of the collaboration relation (i.e. the arcs on the collaboration diagram) should be put in the remote interface. In this way, the methods that can be called locally will not incur the performance penalty produced by the RMI system.

Even if the collaborators can be detected, the issue of *which* local objects should be made remote still remains. As the example in the previous section pointed out, a careful selection of the objects that will be made remote can produce better results. Consider for example the collaboration diagram of Figure 8.5. The arcs are labeled with the expected number of invocations for each particular message. This number can be obtained as a hint from the user or by using an instrumented version of the program (i.e. a version that can record profiling information), and obtaining the numbers by running the program.

Looking at the diagram and without considering the number of invocations, we can say that those objects that have incoming arcs and no outgoing arcs are the best candidates to be made remote because their collaborator set is empty. In our example, moving object **D** to a different virtual machine is easy since no other object has to be made remote. Even though other factors, such as computation granularity

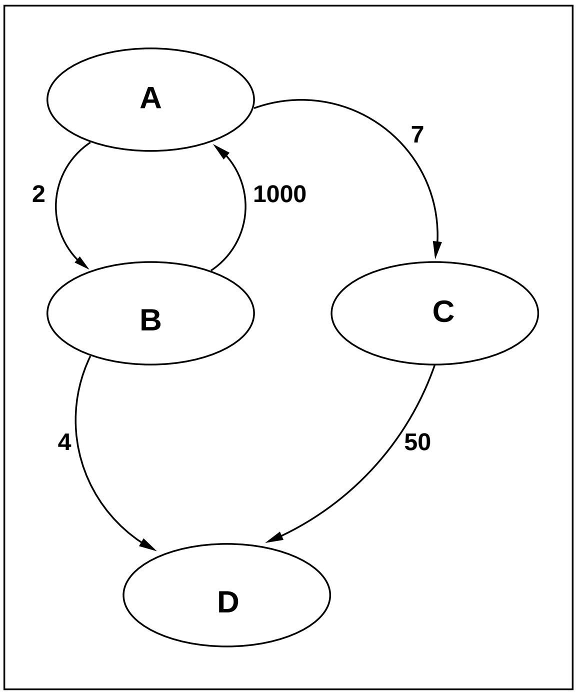

Figure 8.5: Generic Collaboration Diagram

and number of expected invocations, should also be considered before deciding for a particular object, important clues can be gathered from the collaboration diagram. For example, making object **B** remote forces objects **A** and **D** to also be remote. This may be not desired since **B** will incur extra overhead for each of the 1000 invocations of **A**. Before committing to a particular decision though, the computation times of each method should also be considered. It could happen that the computation times are such that moving **B** is the better option. However, if the relative times of the methods involved are similar, the collaboration relations can give a good hint about the partitioning. [2]

The problem of statically assigning tasks to processors has been extensively studied in the past (see [Lo88], [NH85], [TT85] and [WM93]), and those experiences can be applied to address the application partitioning problem (i.e. how to distribute the objects into the available processors).

[2]Note that if an instrumented version of the program is available, after running it the diagram could be extended to include the computation times.

8.4 Summary

Any object whose methods are invoked by another object is said to be a collaborator of the second. As shown in this chapter, converting a local object into a remote object results in all of its collaborators also being converted. If the cost of invoking a remote method were similar to that of a local method, then the collaborators problem could be solved by making everything remote. Unfortunately, this is not true in RMI and, thus, performance will suffer by implementing this solution. To minimize the performance overhead, only some objects and their collaborators should be made remote. The problem of automatically determining all the collaborators of a class requires extra information that cannot be obtained from just a static analysis of the program (e.g. user hints).

Finally, it should be pointed out that the collaborators issue is not a problem of Java or the RMI system. The concept of collaborators is present in any object-oriented language. The issues discussed in this chapter will arise whenever a sequential object-oriented program is translated into a parallel one.

Chapter 9

Runtime System

While some of the RMI's limitations can be addressed by using compiler transformations (e.g. replacing direct field accesses by remote method calls), others, such as the lack of dynamic object creation, require some support infrastructure. This support infrastructure can be provided as a library of classes and a runtime system which can be used as the target for our parallelizing tool. The library of classes will provide the implementation for all the abstractions that will be used by the runtime system and by the parallelizing compiler. For example, a component of this library would be the classes and interfaces that are needed to implement futures as discussed in Chapter 7. The runtime system is a piece of code that is run on every participating node (i.e. each host in the network). When fully implemented, the runtime system will provide the functionality necessary to overcome most of the limitations of the RMI system. The features of the runtime system include:

- Dynamic object creation,
- Remote object's data caching,
- Access to static methods and variables,
- Load balancing and,
- Object migration.

In this chapter we present an architecture for a prototype runtime system that provides dynamic object creation and a basic load balancing scheme. A discussion

```
import java.rmi.*;

public interface ObjectServerRI extends Remote {
    Remote CreateObject(String class_name) throws RemoteException;
}
```

Figure 9.1: ObjectServer's Remote Interface

about how to extend the prototype to include the rest of the features is also presented. Even though our prototype is tailored to Java/RMI, any automatic distribution system (e.g. for C++), will require a similar runtime system (the exact list of features will depend on what it is provided by the underlying system). An example of the transformation of a simple sequential application to a parallel one is also included to illustrate the use of the runtime system.

9.1 Prototype Runtime System

Dynamic object creation is a vital feature for the development of object-oriented parallel applications. Unfortunately, the RMI system does not support it. In RMI, server objects have to be created and started up on different nodes before a client can remotely call a method on them. For translating user sequential code, however, we need to be able to dynamically create objects on different nodes. Thus, one of the responsibilities of the runtime system is to support this feature. For doing so, we will use a remote object, an instance of ObjectServer, on each node that will be responsible for dynamically instantiating remote objects on that node. Each parallel application can have its own set of ObjectServer objects (one per node) or the ObjectServer objects can be shared among several parallel applications. The factors that affect this decision include: fault tolerance, concurrency, and available resources. The remote interface for the ObjectServer is shown in Figure 9.1. A client that has a reference to an ObjectServer can use the method CreateObject() to instantiate a remote object of the specified class on the node where the object server resides. For obtaining a reference to the proper ObjectServer, a Scheduler object is used.

The Scheduler object is responsible for implementing the load balancing policies defined for the application. Figure 9.2 shows the architecture of the runtime system

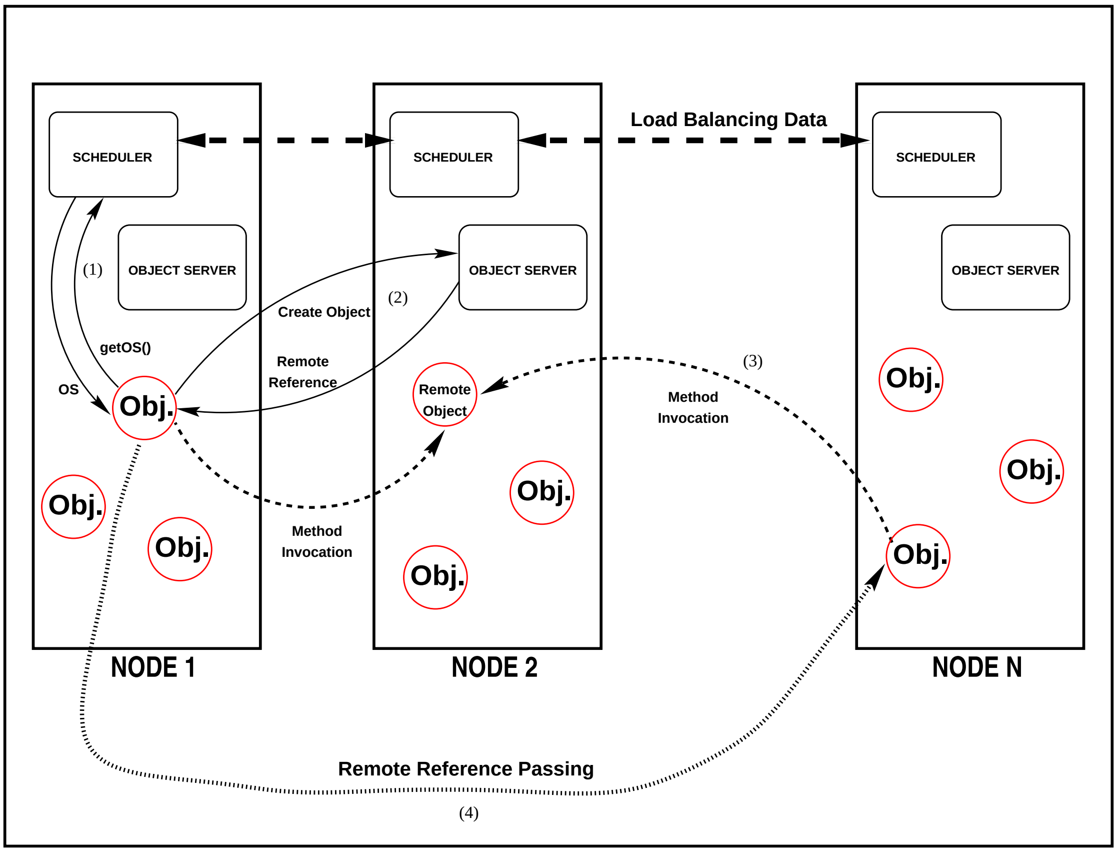

Figure 9.2: The Runtime System Architecture

for an application that is running on **N** nodes. Each node has its own Scheduler and ObjectServer. Each time that an object needs to create a remote object, the local Scheduler is contacted to obtain a reference to the ObjectServer that resides where the object will be created ((1) in Figure 9.2). In the prototype we built, Scheduler objects are regular objects (i.e. not remote) that implement a simple round robin policy among all the participating nodes. The name of a node is obtained from a file that lists all the available nodes. If a more complex load balancing policy needs to be implemented, the Scheduler objects should be made remote so that they can exchange load balancing information among them through remote method invocations.

The ObjectServer implementation creates an instance of the specified class, and exports it before returning the remote reference to the caller (see Figure 9.3). The Init() method registers the ObjectServer object with the local registry (the application name is used as the key). Note that this cannot be done inside the constructor because the object is still being created when executing the constructor and, thus, an extra

```
package RTS;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;
public class ObjectServer extends UnicastRemoteObject implements ObjectServerRI {
    public ObjectServer() throws RemoteException {
      super();
    }
    public Remote CreateObject(String class_name) throws RemoteException {
      Class c;
      Remote r;
      r = null;
      try {
          c = Class.forName(class_name);
          r = (Remote)c.newInstance();
          UnicastRemoteObject.exportObject(r);
      } catch(Throwable e) {RTS.fail("Fatal Error:", e);}
      return r;
    }
    public void Init(String appname) {
        try {
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(appname, this);
        } catch (Exception e) {RTS.fail("OS.Init() failed", e);}
    }
}
```

Figure 9.3: ObjectServer's Implementation

method is needed.

After a remote object is created by the ObjectServer, it can be accessed (i.e. its remote methods executed) directly from any object that possesses a reference to it ((3) in Figure 9.2). Note that a reference to a remote object, if necessary, can be freely passed around through the network ((4) in Figure 9.2).

9.2 Example

In this section we show a complete example of how a sequential application can be modified to run in parallel using our system. The sequential code will be parallelized according to the SPMD model for parallelism, but other models could have been

```
//-[ HellowWorld.java ]-------------
public class HelloWorld {
    static public void main(String arv[]) {
        Hello h = new Hello("World");
        System.out.println(h.Salute());
    }
}

//-[ Hello.java ]-----------------
public class Hello {
    private String msg;
    public Hello(String _msg) {
        msg = _msg;
    }
    public String Salute() {
        return("Hello " + msg + "!!!");
    }
}
```

Figure 9.4: Hello World Application: Sequential Version

chosen instead (i.e. it is not mandatory to run the same program on each node). [1]

Consider the sequential user code shown in Figure 9.4. The HelloWorld application consists of two classes, HelloWorld and Hello. The class HelloWorld implements main() which in turn creates an instance of Hello using the constructor Hello(String _msg). Then it uses this instance to print "Hello World!!!" on the screen. The class Hello provides a method, Salute(), that returns a salutation message.

The first step in transforming this application into a distributed one that uses RMI and the runtime system is to write a remote interface for the class Hello. This remote interface can be easily generated by a compiler and will contain a method declaration for each of the methods in the original class. Figure 9.5 shows the remote interface for the class Hello (i.e. HelloRI). Every constructor is listed as a regular remote method adding void as its return value. This is necessary to remotely invoke the appropriate constructor once the remote object has been created (more on this in the HelloWorld class explanation). Note that RMI requires that the remote methods

[1] In a Single Program Multiple Data (SPMD) model, multiple instances of the same program work on different portions of data. See [Wil95] for a complete taxonomy.

```
//—[ Hello.java ]——————————————————
package HelloWorld;
import java.rmi.*;
public interface HelloRI extends Remote {
    String Salute() throws RemoteException;
    void Hello(String _msg) throws RemoteException;
}
```

Figure 9.5: Hello World Application: Parallel Version, Remote Interface

be public; the compiler should enforce this by changing the declarations as needed.

After completing the remote interface, the implementation of the class Hello should be changed accordingly. Note that the changes are simple and can be easily performed by a compiler (see Figure 9.6). The RemoteException should be added to the throw clause of each method and a no-argument constructor should be provided. [2]

Once the Hello class has been transformed, the HelloWorld application has to be modified. Figure 9.7 shows the modified version of HelloWorld. The application is modified to be executed under the SPMD model. The same code is run on each node; that is the application is separately started up on each node with: java HelloWorld.HelloWorld rank_number. A rank number is added to the application's parameters line so that different nodes can execute different portions of the code (see the rank variable in the code). The RMISecurityManager is made active and an ObjectServer is instantiated on each node. Each node creates a Scheduler to handle the placement of the dynamically created remote objects. Then, the code is split using the rank number. The task with rank "0" (the master task) will be the only one to execute the code inside the "if" statement. The master task will use the Scheduler to obtain a reference to an ObjectServer that is running on some remote node. This reference in turn, is used to create an instance of HelloImpl on a remote node via the CreateObject() method. The appropriate constructor is immediately invoked as a remote method. Note that the modifications to the HelloWorld class are extensive and require significant compiler work. In particular, each method call to an object that has been made remote has to be surrounded by a try/catch clause to handle the remote exceptions that could be generated by the RMI system.

[2]This constructor will be used when the remote ObjectServer instantiates this class.

```
//—[ HelloImpl.java ]————————————————
package HelloWorld;
import java.rmi.*;
public class HelloImpl implements HelloRI {
    private String msg;

    // The no-argument constructor is added
    // for the parallel version (explained in the text)
    public HelloImpl() throws RemoteException {
        super();
    }

    public void Hello(String _msg) throws RemoteException {
        msg = _msg;
    }
    public String Salute() throws RemoteException {
        return(msg);
    }
}
```

Figure 9.6: Hello World Application: Parallel Version, Hello Implementation

9.3 Discussion

The prototype runtime system presented in the previous sections can be modified to support caching of the remote object's data (to increase performance), access to static variables and methods, better load balancing, and object migration.

As explained in Section 6.3, the RMI system does not provide a way for accessing the instance variables of remote objects. The use of remote interfaces as data types prevents the access to the instance variables because variables cannot be defined in an interface. This can be solved (as already proposed) by replacing all data accesses by remote method invocations. The use of these accessor methods, *a priori*, suggests an important performance degradation. To minimize this performance penalty, the runtime system could be expanded to support instance caching. ObjectServer objects could be modified to include a hashtable with cached instance variables from the different remote objects. Different consistency protocols could be implemented (e.g. write-update, write-invalidate) and, at the very minimum, read-only data should be replicated. The code in the get() and set() methods should be modified accordingly

```
//—[ HelloWorld.java ]——————————————
package HelloWorld;
import java.rmi.*;
import RTS.*;
public class HelloWorld {
    public static void main(String argv[]) {
      String rank;
      rank = argv[argv.length - 1];
      System.setSecurityManager(new RMISecurityManager());
      try {
          ObjectServer os = new ObjectServer();
          os.Init("HELLOWORLD");
      }
      catch(RemoteException e) {RTS.fail("OS creation failed", e);}
      Scheduler sched = new Scheduler("HELLOWORLD");
      if (rank.equals("0")) {                              // if Master task...
         try {
            ObjectServer vos = sched.getOS();
            HelloRI h = (HelloRI) vos.CreateObject("HelloWorld.HelloImpl");
            h.Hello("World");
            System.out.println(h.Salute());
         }
         catch(RemoteException e) {RTS.fail("RemoteException", e);}
      }
    }
}
```

Figure 9.7: Hello World Application: Parallel Version

to contact the local ObjectServer for accessing the instance variables. Isenhour [Ise97] proposed a system for doing this that can be adapted to our runtime system.

Another possible use of the runtime system is to provide access to static instance variables and methods. Currently, RMI does not support static instance variables and methods. A possible solution would be to implementing the static parts of a class as a single remote object. When the ObjectServer creates the first instance of a class, it should also create an object to handle the static parts contained in that class. This object will be a remote object whose instance variables and methods are the static instances and methods of the original class respectively. Every object of the original class that is created is given a reference to its static parts. Static methods, can be accessed through this reference as regular remote methods. Access to static instance variables should be changed into the appropriate get() and set() methods on the object that implements the static parts of that class. A similar approach has been taken by JavaParty [PZ97] and their experience can be applied to our runtime system.

The prototype runtime system provides a simple round robin policy for placing the objects on the different nodes. Better load balancing algorithms could be tried instead. This will require that the Scheduler objects be transformed into remote objects and remote methods be provided to exchange load information among the different nodes.

Object migration could also be provided by using the Object Serialization API provided by Java and the ObjectServer objects. To migrate a remote object from one node to a different one, its state should be saved and passed from one ObjectServer object to another. Also, a new exception should be thrown when a remote method call is attempted to a remote object that has been moved to another node. The compiler should insert code to handle this exception and redirect the call to the appropriate node (i.e. the location of the objects should be handled by the runtime system). The code that handles the exception should also update the reference possessed by the client. A similar scheme for object migration has been implemented in JavaParty [PZ97].

All the mentioned extensions to the runtime system will require significant compiler support. The original sequential code has to be extensively modified in order to incorporate all these extensions.

9.4 Summary

In this chapter we presented an architecture for a prototype runtime system to support parallel applications written with RMI. An example was used to show how to modify a sequential program to be executed on several nodes by using the prototype runtime system. Finally, several improvements that can be made to the prototype were discussed.

Chapter 10

Related Work

Several systems exist that can be considered as related work to Java/RMI. Some of these systems, such as CORBA [OMG95] and HORB [Sat96], present an object-oriented distributed model similar to Java/RMI. Also, there are some systems that intend to apply existing parallel programming solutions (such as PVM [Sun90]) for the development of object-oriented parallel applications in Java. Examples of this are JavaPVM [Thu96] and JPVM [Fer96]. Agent-based technology has also been tried in systems like Voyager [VO97]. Finally, there are some systems like JavaParty [PZ97] that extend the functionality of Java/RMI.

It is important to note that since Java is a very active research topic, new systems appear almost every week. Even though the functionality of some these systems overlap with the work presented in this thesis, many of them were not available when we started our project.

In this chapter we discuss these systems in the light of their possible use for parallel programming. The systems are compared against the Java/RMI approach whenever possible.

10.1 CORBA

CORBA [OMG95] is an open standard for application interoperability defined by the Object Management Group (OMG). In the Object Management Architecture proposed by OMG, every piece of software is represented as an object. Objects communicate with each other using an Object Request Broker (ORB). The ORB provides a mechanism by which the objects transparently make requests and receive responses across heterogeneous languages, tools, platforms and networks.

CORBA supports a general Interface Definition Language (IDL) that may be mapped to any implementation language. Interface definitions are specified in IDL and then stored in an interface repository. Clients request services through the ORB by specifying a target object using an object reference. They can inspect interface definitions in the repository to identify objects' services, and the services' request and response formats. The ORB supplies naming services to map (server) object names, marshals call parameters, dispatches requests to service providers, and returns translated service results to the client objects. The ORB interacts with service providers via object adapter skeletons. Language bindings for the OMG's IDL have been specified for C and C++ and bindings for other languages, including Java, are being developed.

Even though CORBA and RMI have many similarities, they differ in some important aspects. The main differences arise from the fact that CORBA was designed to be language and platform independent whereas RMI assumes a uniform Java-only environment. In CORBA, code cannot be moved across machines because the method implementations could be in any binary format. In contrast, RMI makes heavy use of the Java ability to securely download code. Another feature provided by Java/RMI and not present in CORBA is the automatic garbage collection of the remote objects. Also, in RMI, objects can be passed as parameters; CORBA can pass primitive types, references to remote objects, and compositions of these types (structs), but not objects. In addition, RMI can pass and return object types not seen before as long as those types are subtypes of the declared parameter types.

Consider the sample code shown on Figures 10.1 and 10.2. [1] The code in Figure 10.1 defines a generic server that can be used to execute arbitrary computations launched from other nodes. Any class that implements the interface `Compute` can ask the server to execute the computation for it. For example, the code in Figure 10.2 calculates the number `PI` by shipping the computation to a different node.

When the server gets an object of type `PI` as parameter (which it has never seen before), it goes and downloads its class in order to invoke the `run()` method. In CORBA, this cannot be done because the server could potentially be written in any

[1] This code was written by JavaSoft's Ken Arnold and posted on the RMI mailing list.

```
public interface Compute {
    public Object run();
}
public interface ComputeServer extends Remote {
    public Object runIt(Compute computation) throws RemoteException;
}


// The server object invokes the run() method on any Compute
// object that is passed to it.

public class ComputeServerImpl extends UnicastRemoteObject
                                implements ComputeServer {
    public Object runIt(Compute computation) {
        return computation.run();
    }
}
```

Figure 10.1: Computation Server

```
class PI implements Compute {
    private int precision;

    PI(int howManyPlaces) { precision = howManyPlaces; }

    public Object run() {
        double pi = computePIsomehow()
        return new Double(pi);
    }

    public static void main(String[] args) {
        ComputerServer server = getAComputerServer();
        Double pi = server.runIt(new PI(1000));
        System.out.println("PI seems to be " + pi);
    }
}
```

Figure 10.2: PI Computation

language.

Since parallel applications usually are deployed on homogeneous farms of computers, the extra complexity of using CORBA is not justified. This is especially true considering that problems like those of collaborators and exceptions presented in this thesis affect CORBA in the same way as RMI.

10.2 HORB

HORB [Sat96] is a Java ORB (Object Request Broker) whose characteristics and features are very similar to those of RMI. However, the first version of HORB appeared four months before the announcement of RMI.

HORB includes many features that are common with RMI such as:

- Remote method connection,
- Object transfer, and
- Automated garbage collection of remote objects.

Some unique features are also provided such as:

- Dynamic object creation,
- URL based object naming,
- Asynchronous methods,
- Security by distributed access control lists,
- Inheritance support, and
- Persistence facility (with support for object revival).

The HORB package consists of the HORBC compiler, the HORB server, and the HORB class library. This is similar to RMI but with the difference that the HORB system is implemented at the user level. Hence, HORB will run on any platform that has a Java interpreter. [2]

[2]Note that this is true for the RMI system included in version 1.1 but was not the case when RMI was provided as an add-on in version 1.0.2.

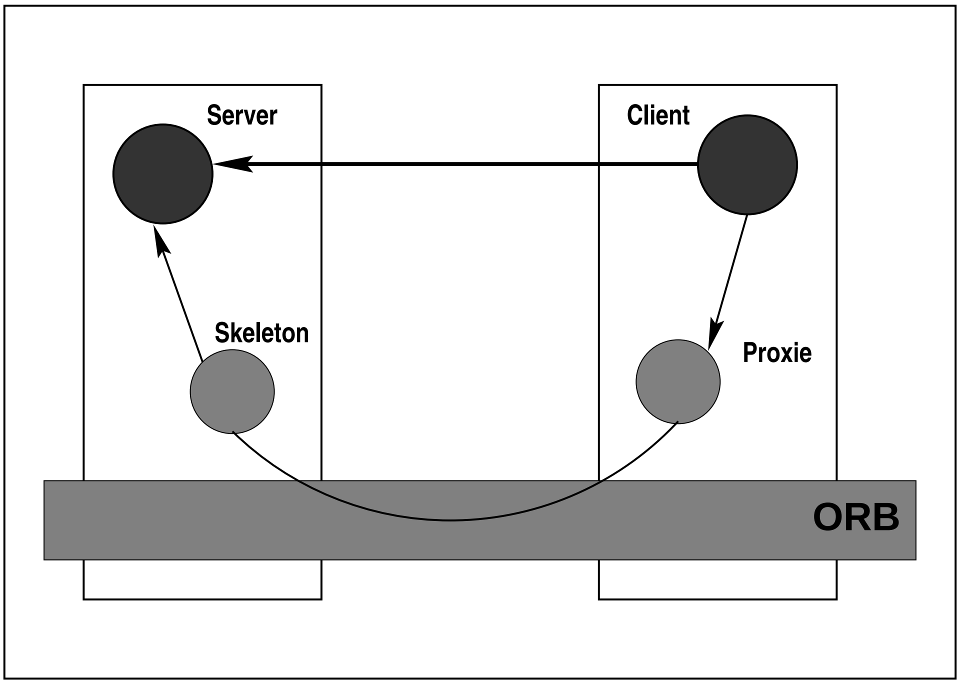

Figure 10.3: HORB Architecture

The HORB architecture is similar to that of RMI (see Figure 10.3). The skeletons and proxies are created by the HORBC compiler and are analogous to the skeletons and stubs on the RMI system. Note that the ORB layer corresponds to the transport and reference layers of RMI. Looking at this, it is clear that the RMI is a more general and extensible system (i.e. it is easier to implement new protocols). However, HORB is stronger than RMI in other aspects.

In general, the HORB system is easier to use because it does not require the use of remote interfaces. Furthermore, a remote class does not have to inherit from a class such as RemoteObject. The stubs and proxies include code for all the methods defined in the class that is being compiled (in RMI only for those that are specified in the remote interface). The inheritance tree of the class is mirrored by corresponding stub and proxy inheritance trees. A remote object is created by directly instantiating the proxie class generated by the HORBC compiler. For example, if there is a class Server (which was compiled with HORBC), a remote instance can be created with the following sentence:

```
Server_Proxy server = Server_Proxy(URL);
```

HORB supports dynamic object creation so the application can decide where to create a particular object (e.g. locally or remotely). In RMI the remote object has to be created by other means (e.g. manually starting up the server) before the client can invoke a method. In HORB, an application can create a remote object in any node just by specifying the URL of the remote node. Moreover, local and remote objects can be integrated by using interfaces. [3] See the following example:

```
interface Server {
     int Compute(...);
}

class Server_Impl implements Server {
   public int Compute(...){...}
}

Server local  = new Server_Impl();   // new local object
Server remote = new Server_Proxy(url); // new remote object
...
local.Compute();
remote.Compute();
```

The code shows an interface declaration, a class that implements that interface (i.e. Server_Impl) and a possible use of that class. Note how an object can be declared of type Server, created locally or remotely, and then used independently of its location.

Another interesting feature offered by HORB and not present in RMI is asynchronous method invocations. For specifying that a method should be invoked asynchronously, the string _Async should be added to the method's name. The asynchronous version of our example server would look something like this:

[3] The use of interfaces is optional in HORB. Its only purpose is for attaining distribution transparency.

```
class Server {
   int Compute_Async(...) {
      ...
      return value;
   }
}
```

For calling the **Compute** method the strings **_Request** and **_Receive** are used in the following form:

```
server.Compute_Request(...);
// continue executing here
value = server.Compute_Receive(timeout);
```

Even though the choice of using special method names seems unnatural, it allows HORB to introduce new semantics without changing the language. These constructs follow the *future* mechanism discussed in Chapter 7. For that reason, the problem that arises when dealing with exceptions and futures also applies to this system (i.e. HORB does not provide anything to handle it differently).

Another interesting feature of HORB is its support for inheritance. In this system, inherited methods can be remotely called without any extra declarations. This constitutes an advance in simplicity over RMI, where the inherited methods have to be explicitly added to the remote interface.

In his paper [Sat96], the author of HORB presents a performance evaluation of his system compared against RMI. It is shown that HORB is two to three times faster than RMI.

Limitations of HORB include the impossibility of directly accessing the fields of remote objects (i.e. accessor methods have to be written as in RMI), and that interface inheritance is not supported (a feature in RMI). Also, the serialization provided in Java is more flexible and complete since HORB does not serialize private fields and does not support class versioning (i.e. objects saved by a class should be restored by the same version of the class).

Overall, HORB is a valid alternative to RMI that provides some features that are missing on RMI. However, some of the problems discussed in this thesis, such as exceptions and collaborators, still apply to HORB.

10.3 JavaPVM and JPVM

The JavaPVM [Thu96] system allows programs written in Java to use the Parallel Virtual Machine (PVM) software developed at Oak Ridge National Laboratory. PVM [Sun90] is a set of software tools and libraries that allows the use of a heterogeneous collection of Unix computers [4] hooked together by a network as a single large parallel computer. Using PVM, the aggregate power of many computers can be combined to solve large computational problems. The source code for PVM is available at no cost. Because of that, PVM has been compiled on almost every architecture. PVM works with programs written in C, C++, and Fortran.

JavaPVM extends the capabilities of PVM to Java, allowing Java applications and existing applications written in C and C++ to communicate with one another using the PVM API. JavaPVM was written using the native method interfaces provided by Java. Essentially, it is a shared library of PVM functions written in C, a Java object class (JavaPVM), and some C code which bridges the gap from the JavaPVM class to the PVM functions. Each function in the PVM API has a corresponding native method declared in the JavaPVM class. For example the standard PVM C function:

```
int tid = pvm_mytid(void)
```

is declared as a method in the JavaPVM class:

```
public native int pvm_mytid()
```

Using the native methods capabilities provided by Java, this JavaPVM method results in a C function:

```
long JavaPVM_pvm_mytid(struct HJavaPVM*)
```

[4] Recently PVM was ported to the Win32 platform (i.e. Windows 95 and Windows NT).

The implementation of this function provided by JavaPVM calls the actual PVM pvm_mytid() function and returns the result. The code for this function looks something like this:

```
long JavaPVM_pvm_mytid(struct HJavaPVM *hpvm) {
  return( (long) pvm_mytid());
}
```

There are some advantages in using JavaPVM over other communication mechanisms. The PVM model for parallel programming is well known and people that are already familiar with it can directly apply their expertise to Java. Also, existing parallel applications written in C, C++ and Fortran can interface with new ones written in Java. This translates into an easier migration path if a language switch is desired.

However, JavaPVM is implemented using the native methods capability of Java. This goes directly against the idea of Java being used as a programming language for heterogeneous environments. [5] Also, the message passing paradigm supported by PVM (and by extension by JavaPVM) lacks a high-level of abstraction. Furthermore, the communication process in PVM is fully exposed to the programmer who is responsible for controlling it (including the marshaling of parameters).

A related system, JPVM [Fer96], is an implementation of the PVM virtual machine written in Java. This system does not use native methods so it fully exploits Java's portability. However, JPVM is not inter-operable with other PVM machines because it is an implementation of the PVM daemon servers rather than an interface between Java and PVM. This seriously restricts the usability of the system. JPVM shares with JavaPVM the same problems regarding abstraction and ease of use.

10.4 Voyager

Voyager [VO97] is an ORB entirely written in Java that allows the development of distributed applications using traditional (i.e. RPC-like) remote method invocations and agent-based distributed programming techniques. Agents are autonomous objects

[5] The native method interface was used to take advantage of the PVM virtual machine already written.

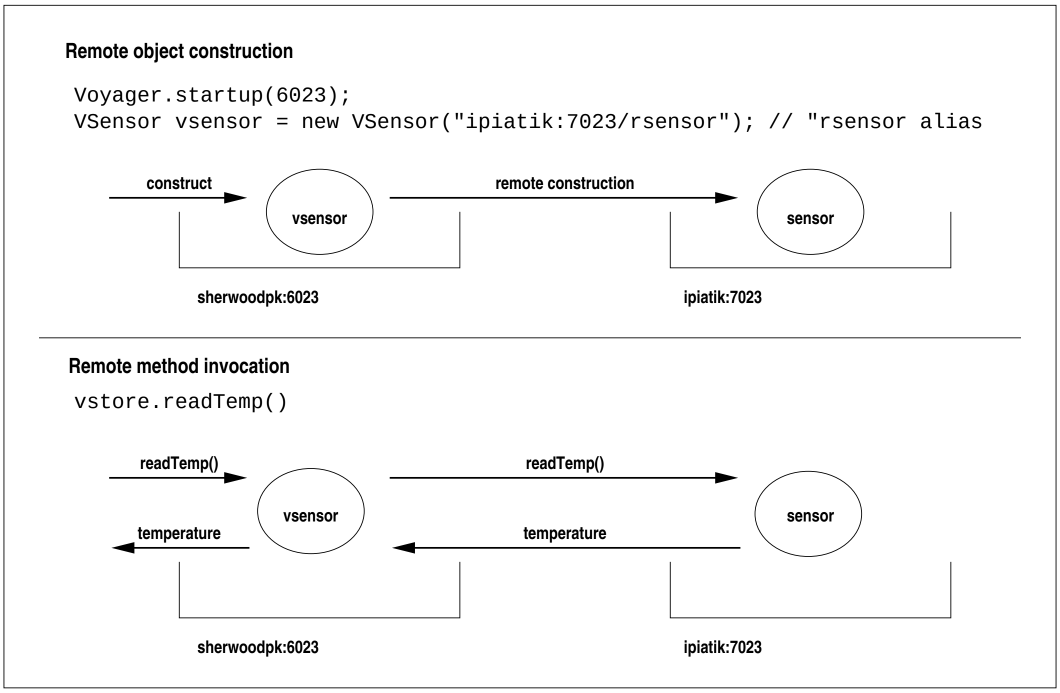

Figure 10.4: Remote Object Creation and Remote Method Invocation in Voyager

that can be programmed to accomplish several tasks. Voyager agents have the ability of moving through the network to accomplish their goals.

Voyager applications contain an infrastructure that allows objects to communicate and move through the network, and several other support services such as distributed garbage collection. Every application is identified by a host name and a port address that is unique to that host (e.g. sherwoodpk:6023). An object residing in one application can communicate with a remote object (i.e. one that resides in a different application) by creating a virtual version of the remote object in the local machine. The virtual version of a remote object is called a virtual object and acts as a *proxy* for the remote object (i.e. the equivalent of a stub in RMI).

Virtual objects are created by instantiating a virtual class. Virtual classes are automatically generated by Voyager from an existing Java class and are named by preceding the original class name with a "V". Voyager only generates code for the client proxies (i.e. there are no skeletons as in RMI), and the virtual class generator (the vc utility) can be used to generate virtual classes from classes for which the source code is not available (i.e. from the .class file). This characteristic allows the programmer to generate remote objects from existing third-party libraries.

Figure 10.4 shows two applications: sherwoodpk:6023 and ipiatik:7023. The code in the upper half of the figure creates a remote object by instantiating the class VSensor (automatically generated by vc from Sensor). The constructor is given the address of an application where the new remote object will be created and an optional alias (rsensor) that can later be used to locate the remote object via the application's registry. The lower half of Figure 10.4 shows how a message, readTemp(), is forwarded from the virtual object to the remote object. Exceptions raised in the remote object are forwarded back to the caller and re-thrown as if they were local exceptions.

In Voyager, messages are *synchronous* by default as in RMI. In addition, however, Voyager supports *one way* messages which return immediately discarding any return value, and *futures* where the call returns immediately with a placeholder that can be used later to retrieve the value. It is important to point out, however, that the future model provided by Voyager is similar to those discussed in Chapter 7. Thus, the use of futures in Voyager will suffer from the same limitations regarding exceptions that we have identified.

Another advantage of Voyager over RMI is that in Voyager objects can be moved from application to application. This is done by simply sending the message move() to the object that needs to be moved, and specifying the destination as a parameter. When an object is moved to another application, it leaves a *secretary* object in its original application. When a message for an object that has been moved arrives to an application, the secretary is contacted to find out the object's new location and the message is automatically forwarded to it. When the object finally receives the message and responds, it includes its new address in the return value so the caller can locate it directly in future requests.

Voyager also supports agent technology. Agents can be created by extending the class Agent and then using the vc utility to create a virtual agent class that, in turn, can be utilized to instantiate an agent object and communicate with it. Agents can also be moved from one application to another but unlike remote objects, agents can move independently (i.e. they can decide where to go when and they move by themselves). An itinerary for an agent can be programmed by specifying an extra call-back parameter in its move() method. For example:

```
public void goShopping() {
  move ("storeOne:3030", "atStoreOne");
}
```

The preceding code specifies that whenever the agent receives the call goShopping(), it will send itself the message move(). It also specifies that after the agent has moved to its new location it will receive the call atStoreOne(). Note that once the agent performs the required computation at its new location, the method atStoreOne() can be used to move the agent to another location (i.e. forming in this way an itinerary).

An agent can also be moved to the location where another object resides. This ability can be used to increase the performance when the agent has to interact heavily with another object. Instead of using remote calls, the agent can be told to move to the same location of the other object and perform all the interactions locally.

Overall, Voyager is a more capable system than RMI. It provides many features like object movement and agents which are not present in RMI. Voyager adds asynchronous messages, although the problem presented by exceptions when combined with futures is not even mentioned in the documentation. Also, the performance of Voyager is much better than that of RMI. See [VR97] for an exhaustive comparison between Voyager an RMI.

10.5 JavaParty

JavaParty [PZ97] is a system that targets the development of object-oriented parallel applications running on clusters of workstations using Java and RMI. The programmer turns a multi-threaded Java program into a JavaParty program by indicating which classes and threads should be made remote. This is done by using a new modifier **remote** added to the language.

Figure 10.5 shows how a JavaParty program is translated into Java bytecodes. The code is first run through a compiler that translates the JavaParty code (i.e. the remote declarations) into Java/RMI code. Then the normal Java and RMI compilers are used to obtain the final bytecodes.

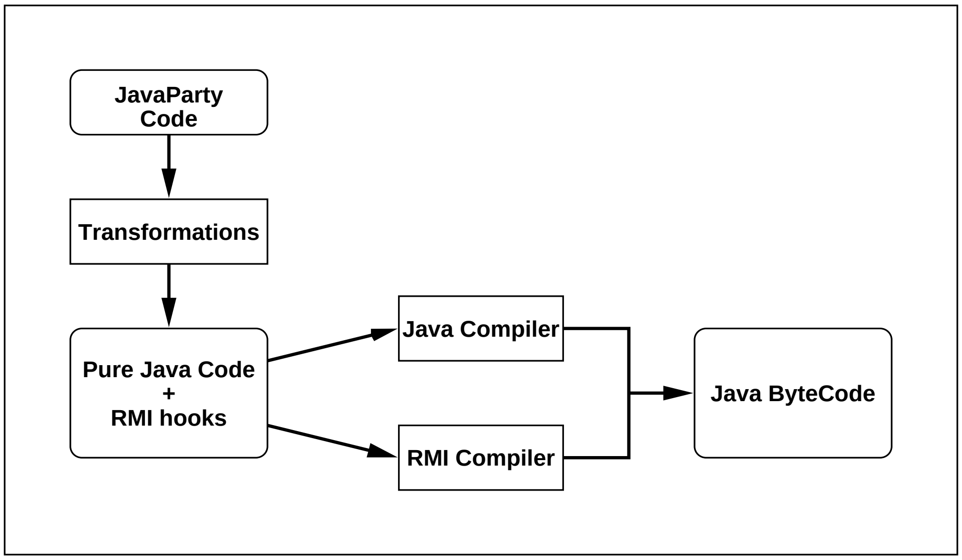

Figure 10.5: JavaParty Code Translation

JavaParty provides several advantages over the use of RMI alone. It abstracts the programmer from the complexities of RMI (e.g. declaring methods in the remote interfaces and catching extra exceptions). As a consequence, the code written by the user when using JavaParty is shorter than the equivalent RMI version (although the code generated by JavaParty is much longer).

JavaParty also provides object migration (although, in fact, objects can only be moved when they are not executing any method), and avoids the RMI overhead when objects reside on the same virtual machine (which is a very important performance consideration). Another added feature in JavaParty is the ability to access class variables and methods (i.e. static fields and static methods). In RMI, these static parts cannot be accessed.

Besides the compiler, the JavaParty system includes a runtime system that is used to support access to the static parts of the classes and the migration of objects. This runtime system consists of a central component that is unique to the system and a local manager per node. The central manager knows all the local managers and the location of all the objects that implement the static parts of a class. All this information is replicated in the local managers to reduce the load on the central manager. Objects are created according to distribution strategies that can be selected and changed at runtime.

JavaParty is a good alternative to Java/RMI alone. In fact, it solves some of the problems of RMI and provides an easier model to write parallel applications. Even though some of the problems of RMI still remain, JavaParty constitutes a step forward over RMI. The ability to access the static parts of a class is an important feature for the development of automatic parallelization tools since class variables and methods are present in almost every sequential program. Also, given the performance of the RMI system (see Chapter 5), the optimization of message invocations between remote objects that reside on the same machine is extremely valuable. Finally, the object migration facility provided by JavaParty provides a more flexible distribution model (i.e. the placement of the objects can be dynamically changed).

10.6 Summary

In this chapter, some approaches to object-oriented parallel applications were discussed. CORBA and HORB were described as having models of object distribution similar to RMI. JavaPVM and JPVM were presented as examples of a different approach that is closer to socket programming than to object distribution. Agent-based approaches were illustrated with the Voyager system. Finally, JavaParty was shown as a system that is based on the Java/RMI model but that extends that model to include some important features for parallel programming.

Chapter 11

Conclusions

This thesis is an evaluation of Java and RMI for the development of object-oriented parallel applications. In particular, the thesis looks at the issues involved in automatically distributing sequential object-oriented programs by using a parallelizing tool. The focus is placed on how the features of the language, the object-oriented technology used, and the parallelization techniques applied can be combined for the development of distributed applications.

While some of the issues raised in this thesis can be directly related to the Java/RMI design, others are intrinsic to the object-oriented paradigm used in the programs, and some others to the parallel computing model.

The problems directly related to Java/RMI identified in this thesis include:

Lack of transparency: The RMI system forces the programmer to declare which methods can be remotely called by the use of remote interfaces (including the class' methods and any inherited methods as well). Also, classes are required to implement the Serializable interface so that their objects could be used as parameters or return values of remote methods. Furthermore, all the remote methods should be public. As shown in this thesis, this lack of transparency of the RMI system can be overcome by the use of extensive compiler transformations (to adapt the user's code to the RMI syntax), and by some changes in the implementation of the RMI system itself (e.g. the elimination of the Serializable requirement).

Performance: The RMI system proved to be a major bottleneck. The performance evaluation in Chapter 5 showed the huge overhead incurred by the applications when using RMI. This overhead is caused by the complexity of the different layers of the RMI system (i.e. stubs/skeletons, reference and transport). Furthermore, a decompilation of these layers also revealed the presence of debugging code. The serialization routines, even though they can be optimized, were not as inefficient as RMI itself. It is clear that a method must have a large granularity to offset the overhead of the RMI system.

Access to instance variables: RMI does not provide a way of accessing the instance variables of the remote objects. Compiler transformations could be used to overcome this limitation. As discussed in Section 6.3, a solution is to change all the accesses to instance variables into calls to appropriate `get()` and `set()` remote methods.

Access to static instance variables and methods: As pointed out in Chapter 3, `static` instance variables and methods cannot be used with RMI alone. This happens because clients only see what is declared in the remote interface of an object and the `static` keyword is not allowed inside interfaces. A runtime system is needed for supporting access to remote `static` variables and methods.

Even though these problems affect the automatic parallelization of the user's sequential code, they can be solved by using compiler transformations and a runtime system. A prototype system that addresses many of these concerns was described in Chapter 9. The performance overhead, however, constitutes a very strong factor against the use of RMI as the target of our parallelization tool. If RMI is used directly by the user, the performance degradation is partially compensated by its ease of use. However, when using a parallelizing tool, it is preferable to generate the fastest code that is possible because the ease of use will already be provided by the environment, that is, the user will write sequential programs.

Regardless of the communication mechanism used (e.g. RMI, sockets, etc.), this thesis identified some harder problems which are not caused by Java/RMI such as:

Exceptions: As demonstrated in Chapter 7, the exception handling mechanisms included in some modern languages, such as Java and C++, can seriously limit the amount of concurrency in a parallel application. Although we presented some approaches for dealing with this problem, further research will be necessary to find solutions that provide better performance.

Collaborators: Chapter 8 showed the problems that any object-oriented language with dynamic dispatching can present for detecting the collaborator set. This issue requires further work for the development of a tool to automatically determine the collaborator set.

The only problem related to the parallel computing model is how to stop a running computation in a speculative computing environment where several futures are created to find or compute the same value. As shown in Chapter 7 this can be solved by providing an extra method in the server or in the future (according to which implementation of futures is used) to stop the appropriate thread. This problem does not constitute an important impediment for the development of a parallelization tool as its solution is straightforward.

Because of the problems explained above, we conclude that any automatic parallelizing tool for an object-oriented language will have to deal with the issues of exceptions and collaborators. RMI, in addition, presents many problems that can be addressed as discussed in this thesis. However, given its performance, we also conclude that the amount of compiler support needed to address all these problems is too large compared with the benefits obtained by using the RMI system. Even though RMI can be successfully applied to other domains, such as distributed client/server databases where the performance overhead of the RMI system is not a predominant factor, it is unsuitable for being used as the underlying communication paradigm of our parallelizing tool.

Bibliography

[Ame87] Pierre America. "POOL-T: A Parallel Object-Oriented Language". Object-Oriented Concurrent Programming, MIT Press, 1987.

[AH87] G. Agha and C. E. Hewitt. "Concurrent Programming Using Actors". Object-Oriented Concurrent Programming, MIT Press, 1987.

[BD+92] P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younge, and C.R. Zarnke. "μC++: Concurrency in the Object-Oriented Language C++". Software: Practice and Experience, vol. 22, no. 2, pp. 137-172, 1992.

[BD+93] Adam Beguelin, Jack Dongarra, Al Giest, Robert Mancheck, and Keith Moore. "HeNCE: A Heterogeneous Network Computing Environment". Technical Report UT-CS-93-205, University of Tennessee, August 1993.

[BH77] Henry Baker Jr. and Carl Hewitt. "The Incremental Garbage Collection of Processes". Proceedings of the Symposium on Artificial Intelligence and Programming Languages, pp. 58-59, August 1977.

[BN83] A. D. Birrel and B. J. Nelson. "Implementing Remote Procedure Calls". XEROX CSL-83-7, October 1983.

[Cox86] B.J. Cox. "Object Oriented Programming - An Evolutionary Approach". Addison-Wesley, 1986.

[Eng97] Jason English. "It all started with a blunt letter". Java FAQ, Sun Microsystems Inc., 1997. http://www.javasoft.com/nav/whatis/index.html

[Fer96] Adam Ferrari. "JPVM" http://www.cs.virginia.edu/~ajf2j/jpvm.html

[Fla96] David Flanagan. "Java in a Nutshell". O'Reilly & Associates, Inc. February 1996, First Edition.

[Gri93] A. S. Grimshaw. "Easy to Use Object-Oriented Parallel Programming with Mentat". IEEE Computer, vol. 26, no. 5, pp. 39-51, May 1993.

[GH+94] E. Gamma, R. Helm, R. Johnson and J. Vlissides. "Design Patterns: Elements of Reusable Object–Oriented Software". Addison–Wesley, Reading, Massachusetts, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. "The Java Language Specification". Addisson-Wesley Java Series, September 1996.

[GM96] James Gosling and Henry McGilton. "The Java Language Environment: A White Paper". Java White Papers, Sun Microsystems Inc., 1996. http://www.javasoft.com/nav/read/whitepapers.html

[GR83] A. Goldberg and D. Robson. "Smalltalk 80: The Language and its Implementation". Addison-Wesley, May 1983.

[Hal85] Robert Halstead. "Multilisp: A Language for Symbolic Computation". ACM Transactions on Programming Languages and Systems, October 1985.

[Ham96] Marc A. Hamilton. "Java and the Shift to Net-Centric Computing". Computer, pp. 31-39, August 1996.

[HCU91] U. Holzle, C. Chambers and D. Ungar. "Optimizing Dynamically-Typed Object Oriented Languages with Polymorphic Inline Caches". ECOOP 91, Conference Proceedings, 1991.

[Ise97] Philip Isenhour. "Property Caching for Remote Java Objects". http://simon.cs.vt.edu/ isenhour/rmicache/

[JO95] "The Java Language: An Overview". Java White Papers, Sun Microsystems, Inc., 1995. http://www.javasoft.com/nav/read/whitepapers.html

[Kra96] Douglas Kramer. "The Java Platform: A White Paper". Java White Papers, Sun Microsystems, Inc., 1996.
http://www.javasoft.com/nav/read/whitepapers.html

[Lo88] V. M. Lo. "Heuristic Algorithm for Task Assignment in Distributed Systems". IEEE Transactions on Computers, vol. 37, no. 11, November 1988.

[MPI94] Message Passing Interface Forum. "MPI: A Message-Passing Interface Standard". May 5, 1994. University of Tennessee, Knoxville, Report No. CS-94-230. See also: International Journal of Supercomputing Applications, vol. 8, no. 3/4, 1994.

[Nie87] Oscar Nierstraz. "Active Objects in Hybrid". OOSPLA '87 Proceedings, ACM SIGPLAN Notices, vol. 22, no. 12, pp. 243-253, December. 1987.

[NH85] L. M. Ni and K. Hwang. "Optimal Load Balancing in a Multiprocessor System with Many Job Classes". IEEE Transactions on Software Engineering, vol. SE-11, no. 5, May 1985.

[OMG95] Object Management Group. "The Common Object Request Broker: Architecture and Specification". Revision 2.0, July 1995.

[OS96] "Java Object Serialization Specification". Sun Microsystems, Inc., 1996.

[PZ97] Michael Philippsen and Matthias Zenger. "JavaParty - Transparent Remote Objects in Java". University of Karlsruhe, Germany.
http://wwwipd.ira.uka.de/~phlipp/party.ps.gz

[RMI97] "Java Remote Method Invocation Specification". Revision 1.4, JDK 1.1 FCS, February 10, 1997. Sun Microsystems, Inc., 1997.

[Sun90] V. Sunderam. "PVM: A Framework for Parallel Distributed Computing". Concurrency: Practice and Experience, 2(4):315-339, December 1990.

[Str91] Bjarne Stroustrup. "The C++ Programming Language". Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[Sat96] Hirano Satoshi. "HORB: Distributed Execution of Java Programs". http://ring.etl.go.jp/openlab/horb/

[SS96] Duane Szafron and Jonathan Schaeffer. "An Experiment to Measure the Usability of Parallel Programming Systems". Concurrency: Practice and Experience, vol. 8, no. 2, pp. 147-166, 1996.

[SS+93] Jonathan Schaeffer, Duane Szafron, Greg Lobe and Ian Parsons. "The Enterprise Model for Developing Distributed Applications". IEEE Parallel and Distributed Technology, 1(3):85-96, 1993.

[Thu96] David A. Thurman. "JavaPVM v1.0.1" http://homer.isye.gatech.edu/chmsr/JavaPVM/

[TT85] A. N. Tantawi and D. Towsley. "Optimal Static Load Balancing in Distributed Computer Systems". Journal of the ACM, vol. 32, pp. 445-465, April 1985.

[VO97] "Voyager Core Package Technical Overview". Voyager White Papers, ObjectSpace, Inc., 1997.
http://www.objectspace.com/Voyager/voyager_white_papers.html

[VR97] "Voyager and RMI Comparison". Voyager White Papers, ObjectSpace, Inc., 1997.
http://www.objectspace.com/Voyager/voyager_white_papers.html

[Weg87] Peter Wegner. "Dimensions of Object-Based Language Design". Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), in SIGPLAN Notices vol. 22, no. 12, pp. 168-182, December 1987.

[Wil95] Gregory V. Wilson. "Practical Parallel Programming". MIT Press, Scientific and Engineering Computation Series, 1995.

[WM93] Murray Woodside and Gerald Monforton. "Fast Allocation of Process in Distributed and Parallel Systems". IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 2, February 1993.

[YT87] Y. Yokote and M. Tokoro. "Concurrent Programming in ConcurrentSmalltalk". Object-Oriented Concurrent Programming, MIT Press, 1987.