Journal of Universal Computer Science, vol. 9, no. 8 (2003), 761-775
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © J.UCS

Asynchronous Remote Method Invocation in Java

Wendell Figueiredo Taveira
(Computer Science Department, Federal University of Minas Gerais, Brazil
taveira@dcc.ufmg.br)

Marco Tulio de Oliveira Valente
(Computer Science Department, Catholic University of Minas Gerais, Brazil
mtov@pucminas.br)

Mariza Andrade da Silva Bigonha
(Computer Science Department, Federal University of Minas Gerais, Brazil
mariza@dcc.ufmg.br)

Roberto da Silva Bigonha
(Computer Science Department, Federal University of Minas Gerais, Brazil
bigonha@dcc.ufmg.br)

Abstract: Java RMI is the computational model used to develop distributed systems in the Java language. Although widely used in the construction of distributed systems, the use of Java RMI is limited because this middleware does not allow asynchronous method invocations. This paper presents FlexRMI, a Java based system that supports asynchronous invocations of remote methods. FlexRMI is completely implemented in Java, making use of the reflection and dynamic proxy facilities of this language. The implementation is also compatible with standard Java RMI distributed systems.

Key Words: Distributed programming, asynchronous remote method invocation.
Category: D.1.5, D.1.3, D.3.3

1 Introduction

Distributed system notion has progressively been established as the standard platform for software development. Differently from centralized systems, distributed systems are designed as a set of autonomous processes interconnected in a network. Each process is called node. This kind of environment is well characterized in the Internet, where heterogeneous processors, probably located in different places, comprises a huge computer network [5].

Distributed object systems, such as CORBA [9] and Java RMI (*Remote Method Invocation*) [1, 12], traditionally offer only a single communication mechanism between nodes, the synchronous communications. Synchronous invocations are indeed quite adequate to local network environments, where some of its characteristics are:

- previsibility;

– latency with upper bound defined;

– reliable and steady bandwidth.

However, framework based on exclusively synchronous communication is not adequate to the modern network environments, such as Internet, mobile computation, and wireless network. These new networks are characterized by non steady bandwidth and by high latency. Consequently, it is hard to set an upper bound on the delay needed to send and receive messages, which could be used to recognize communication failures. Note that, in local network, the usual delay upper bound in the order of milliseconds may give rise to systems with low degree of fault tolerance.On the other hand, an upper bound limit in the order of seconds may cause an intolerable degradation on the applications turnaround time. In this case, overlapped computation and asynchronous execution of services might be in order [14].

Nowadays it is common practice to incorporate asynchronous communication mechanism to distributed systems. The latest version of Corba, in recognition of these problems, embodies synchronous methods invocations. Java RMI, however, still does not has this functionality. The computational model of Java RMI allows the invocation of methods on specified remote objects. The syntax of such a remote invocation is similar to a local invocation, what makes it easy to use. Java RMI may be used in the development of pure Java system. Besides that, Java RMI forms the basis of new technology such as Jini [15], a *middleware* adequate to reconfigurable networks.

The goal of this work is to investigate the development of a distributed object system that allows synchronous and asynchronous remote invocation of methods. We propose a compatible extension of Java RMI to support asynchronous invocation, called FlexRMI. Our system was designed and implemented taking Java RMI as its starting point.

FlexRMI is then a hybrid model allowing both asynchronous or synchronous remote methods invocations. There is no restrictions in the ways a method is invocated in a program. The same method can be called asynchronously at one point and synchronously at another point in the same application. It is the programmers responsability the decision on how the call is to be made.

The remain sections of the paper is organized as follows: Section 2 briefly presents an overview of the state of art considering the available environments which support asynchronous calls of remote methods. Section 3 describes the RMI model of Java. Section 4 provides a few guidelines to implement programs using the FlexRMI. Section 5 presents important details necessary to understand the implementation of FlexRMI. Section 6 deals with FlexRMI drawbacks and advantages. Finally, Section 7 draws conclusions and directions for further works on this subject.

2 State of the Art

The construction of environment for asynchronous invocation of remote methods has been explored in several work found in the literature.

In Multilisp [4], it is showed how to incorporate parallelism into programs, by means of the `future` mechanism, which gives the idea of `future contract`.

[8] describes the design of a new data type called `promises` that was influenced by the `future` mechanism of MultiLisp. Like `futures`, `promises` allow the result of a call to be picked up later in the program. `Promises` are strongly typed and thus avoid the need for runtime checking They also allow exceptions to be propagated in a convenient manner.

[10] presents a system built on top of RMI that allows concurrent execution of local and remote computations. The system uses the concept of a *future*, which means, when the client calls an asynchronous remote method, that the method returns an instance of the class *Future*, created by a special *stub* object. Besides other functions, the `future` is used by the server to send exceptions to the clients.

In [7], an A-RMI (*Active Remote Method Invocation*) can reference and synchronously or asynchronously remote invoke its public methods using the same syntax as if it were a local object. Three key abstractions provided by the model are: i) the active remote objects with user level scheduling; ii) asynchronous method invocation with data-driven, non-blocking synchronization using call-handlers; and, iii) transparent remote object creation.

[3] presents an extension of standard Java RMI implementation by specializing the stub generation process to include additional communications protocols. To indicate which methods may use asynchronous communication and future objects they have defined a pseudo Java interface with the keywords `asynch` and `future`. The proposed extended `stub` mechanism causes a clear performance increase by removing unnecessary delays caused by the blocking mechanism of synchronous RMI invocations.

Every work mentioned above proposes an asynchronous remote method invocation mechanism and present solutions to improve performance, mostly in synchronization situations.

Besides Java RMI, other works propose solutions to the problem of asynchronous communication in different technologies.In [11], it is presented a framework whose model is used in CORBA. The approach shows how to implement C++ programs using the models of *polling* and *callback* to asynchronously invoke remote methods.

All the works presented in this section requires some alterations into Java language or on its compiler which computes the *stubs* and *skeletons*. Our approach, whose results are presented in Sections 4, 5 and 6, provides a solution to overcome these problems.

3 Java RMI

Java RMI [12, 16, 2, 13] is a model of distributed object system for Java environment. Specifically, Java RMI allows developers to treat remote objects in the same way local objects are addressed, encapsulating all their implementation details. Generally, to work with Java RMI, all needed is to import one package, look up the remote object in the registry file and to assure that the `RemoteExceptions` are caught when the method is invoked on the remote objet.

In the environment of Java distributed object model, a remote object is the one whose methods can be invoked from another Java Virtual Machine (JVM), and, even, from different host. Remote objects are defined through one or more remote interfaces. The interfaces are responsible to assure type consistency between the client and the server models.

The interfaces and classes responsible for specifying the remote behavior of the RMI system are defined in `java.rmi` and `java.rmi.server` packages. Figure 1 shows the relationship between these interfaces and classes.

The Java RMI introduces a new object model that extends the object model used up to JDK 1.1, the Distributed Object Model of Java. What follows is a brief comparison of the distributed object model and the Java object model [13, 16]. The similarities between these models are:

- a reference to a remote object can be passed as an argument to method or returned as a result of the method invocation, regardless of whether it is local or remote;
- a remote object can be cast to any set of remote interfaces supported by the implementation using the built-in Java syntax for casting;
- the built-in Java `instanceof` operator can be used to test the remote interfaces supported by a remote object.

The basic differences between the two models are:

- clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces;
- clients must handle an additional exception for each remote method invocation. This is important because the failure modes of invoking remote objects are more complicated than the modes of invoking local objects;

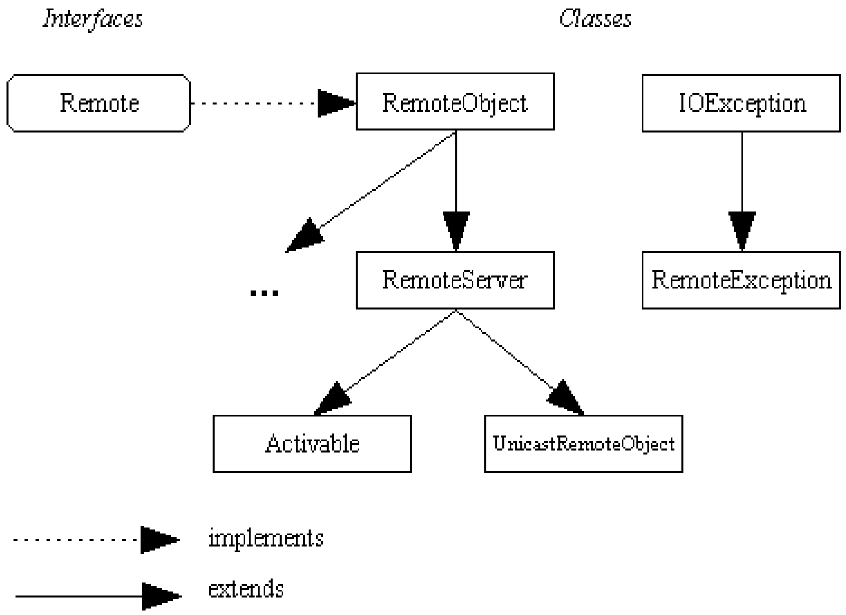

Figure 1: Relationship between Java RMI Interfaces and Classes

- semantics of parameter passing are slightly different in calls to remote objects. Particularly, Java RMI uses the object serialization service, which converts Java objects into a serial stream so that the object state is preserved and can be recovered after been transmitted in the network;
- semantics of Object methods are defined according to remote object models;
- extra security mechanisms are introduced to control the activities that a remote object can perform on a system. A Security Manager object needs to be installed to check all operations of the remote object on the server.

3.1 Architecture of Java RMI System

In the model of distributed objects of Java [16, 6], a client object never references directly a remote object. It must reference a remote interface which is implemented by the remote object. The use of remote interface allows server objects to differentiate between their local and remote interfaces. For instance, an object could provide methods to objects that run in the same JVM, in addition to methods that would be provided by remote interface. The use of remote interfaces also allows server objects to present different modes of remote access. For example, a server object may provide different interfaces to different group of users. Lastly, the use of remote interfaces allows that the position of the server objects into their class hierarchy be abstracted, such that, client objects may be compiled using only the remote interface, independently of the server classes.

4 FlexRMI Description

This section presents a small user guide for the FlexRMI system, which is a mechanism to invoke methods of objects that are in another address space, potentially on a different machine. The most important advantage of FlexRMI over Java RMI is the possibility to invoke methods that will run asynchronously.

4.1 FlexRMI Use

Basically there are three elements that must be considered to carry out a remote method invocation:

1. The **client** which is the process that invokes a method of a remote object.
2. The **server** which is the process that owns the remote object. Remote object is a object whose class extends the class `UnicastRemoteObject` and executes in the address space of the server process.
3. The **Object Register** which is a process that relates objects with names. Objects are registered within the Object Register. Once objects are registered, one can obtain access to remote objects using the name of the objects.

There are two kinds of classes that can be used in FlexRMI:

1. A remote class which is one whose methods can be used remotely. A remote class extends the class `UnicastRemoteObject` and implements an interface that extends the `Remote` interface. An object instance of such a class can be referenced in two different ways:
   - 1.1. In the address space where the object was constructed, and in which it can be used like any other object.
   - 1.2. In other address spaces in which the object can be referenced using the object handler. Asynchronous calls always return an object handler, which is a value of type `Promises`, which permits the invocation state inspection. Details of class `Promises` will be presented in Section 4.1.1.

Although there are limitations on how one can use an object handler compared to an object, for the most part one can use object handlers in the same way as an ordinary object. If a remote object is passed as a parameter to a method or returned as its value, then the object handler is copied from a space address to another one.

2. A serializable class is one whose instances can be copied from one address space to another. An instance of a serializable class will be called a serializable object. If a serializable object is passed as a parameter or returned as a value of a remote method invocation, then the serialized value of the object will be copied from one address space to the other.

4.1.1 The FlexRMI Package

This section presents the description of the classes `Promises` and `FlexRMI` that compose the FlexRMI package. The `Promises` class allows users to access the object handler used in asynchronous invocation and the `FlexRMI` class controls the initialization process necessary to make FlexRMI work. The remaining classes of the package are not directly used by the users in the development of applications that uses FlexRMI.

The class `FlexRMI` has the following method:

– `public static Object lookup (String name)`: this method is used by the application clients to find the remote object `name`. It corresponds to method `Naming.lookup(String name)` of Java RMI.

The methods of the class `Promises` are:

– `public boolean isAvailable()`: returns `true` if the value of the object handler is available. Otherwise, returns `false`.

– `public Object getResult()`: returns the value of the handler object. If the value is not available, the execution is blocked until the value becomes available.

– `public void setResult (Object result)`: assigns the parameter value to the object handler.

4.2 Remote Classes and Interfaces

This section shows how to define a remote class. A remote class has two parts: the interface and the class itself. The remote interface must have the following properties:

1. The interface must be public.
2. The interface must extend the `java.rmi.Remote`.
3. Every method in the interface must declare that it may throws `java.rmi.RemoteException`. This exception is thrown in case some communication fault happens with the server while processing a call. Other exceptions may also be thrown.

The remote class must have the following properties:

1. It must implement a remote interface.
2. It must extend the `java.rmi.server.UnicastRemoteObject`. Objects of such a class are in the address space of the server and its methods can be invoked remotely.
3. It may have methods that are not in its remote interface. In this case, these methods can only be invoked locally.

Unlike the case of serializable class discussed in Section 4.2.1, it is not necessary that the client and server have access to the definition of the remote class. The server requires the definition of both the remote class and the remote interface, but the client only uses the remote interface. In other words, remote interface represents the type of an object handler, while the remote class represents the type of an object. If a remote object is being used remotely, its type must be declared to be the type of the remote interface, not the type of remote class.

4.2.1 Serializable Classes

A class is serializable if it implements the interface `java.io.Serializable`. Subclasses of a serializable class is also called serializable. Details of these kind of classes can be found in [1].

To use a serializable object in a remote method invocation is simple. One simply passes the object as a method parameter or receives it as the method return value. The client and server programs must have access to the definition of the serializable classes that have been used. If the client and server programs are on different machines, then the definition of serializable classes have to be download from one machine into the other.

4.2.2 Programming the Server

In the program showed below, there are a remote class and its corresponding remote interface, that are called, respectively, `Alo` and `AloInterface`. The file `AloInterface.java` contains:

```
import java.rmi.*;
public interface AloInterface extends Remote {
   public Promises say() throws RemoteException;
}
```

The file `Alo.java` contains:

```
01 import java.rmi.*;
02 import java.rmi.server.*;
03 public class Alo extends UnicastRemoteObject
                    implements AloInterface {
04    private String mensagem;
05    public Alo (String msg) throws RemoteException {
06       mensagem = msg;
07    }
08    public Promises say() throws RemoteException {
09       Promises h = new Promises();
10       h.setResult(message);
11       return h;
12    }
13 }
```

It is important to observe that the method `say()` will be called asynchronously, because its return value was declared as type `Promises`. This is showed in line 08 of code class `Alo`.

At this point, the compiler `flexrmic` plays a very important rôle. Suppose that in the server there exist methods whose types of return values are distinct from `Promises`. Up to now, it were not possible to invoke these methods asynchronous. Nevertheless, it is through the `flexrmic` compiler that this limitation is taken care of, making that the choice between a synchronous or asynchronous invocation become a client decision, i.e., who demands the service, and not from the one that provides it. To accomplish this, the compiler flexrmic prefixes the method name that will be invoked asynchronous with the keyword `async_`.

On the server side, there must be a class to instantiate the remote objects. This class does not need to be remote or serializable, although the server uses these class. At least one remote object must be registered in the Object Register. The way to do it is: `Naming.rebind (objectName, object)`, where `object` is the remote object to be registered and `objectName` is a *string* that names the remote object. An example of this class is:

```
import java.rmi.*;
import java.rmi.server.*;
public class AloServidor {
   public static void main (String[] argv) {
      try {
         Naming.rebind ("Alo", new Alo ("Hello, world"));
         System.out.println ("Server Alo ready!");
      }
      catch (Exception e) {
```

```
         System.out.println ("Failure on Server Alo: " + e);
      }
   }
}
```

The Object Register, called `rmiregistry`, only accepts requests to bind and unbind objects running on the same machine, so it is never necessary to specify the name of the machine when one is registering an object.

The code for the server process can be placed in any convenient class. In the example of Server, it was placed into the class `AloServidor` that contains only the program above.

All the classes and the interfaces must be compiled using the `javac`. The file with the remote interface should be compiled with the `flexrmic` compiler, available in the FlexRMI package. Once compiled, the files *stubs* and *skeletons* must be generated using the *stub* compiler called `rmic`. The *stub* and the *skeleton* of the remote interface from the example are compiled by means of the statement: `rmic Alo`.

4.2.3 Programming the Client

The client itself is also a Java program. The invocation of a remote method can return a remote object as its return value. The name of a remote object includes the following informations:

1. The name or address of the machine which runs the Object Register with which the remote object is being registered. If the Object Register is running on the same machine as the one that is making the request, then the name of the machine may be omitted.

2. The communication port to which the Object Register is waiting for requests. The standard port is 1099. In case the standard port is in use, then this does not have to be included in the name.

3. The local name of the remote object within the Object Register.

The client program is:

```
public class AloCliente {
   public static void main (String[] argv) {
      try {
         AloInterface alo = (AloInterface) FlexRMI.lookup ("Alo");
         Promises h = alo.say();
         /*
            Execute other tasks. The status of the object Promisses
            may be inspect at any time via operator isAvailable().
         */

```

```
          h.getResult();
       }
       catch (Exception e) {
       System.out.println ("Exception raised in AloCliente: " + e);
       }
    }
}
```

The `FlexRMI.lookup` method obtains an object handler from the Object Register running on its own machine, since no information about the machine address has been given. It uses the default communication port. The result `FlexRMI.lookup` must be cast to the type of the remote interface.

The remote method invocation in the example client occurs with `alo.say()`, and it immediately returns an object of type `Promises`, which is serializable and the execution of the client continues as usual. When the execution of the called method is finished, its result is deposited in the `Promises` object. So, it is necessary that the client test, via method `isAvailable`, if the value is already available in the `Promises` variable before try to access it. This is made by the `getResult()` method and this point of program works as a synchronization point.

The client code can be in any class. In this example, it is in the `AloCliente` class, which contains only the program above.

4.2.4 Starting the Server

Before starting the server, it is necessary to start the Object Register and leave it running in the background. This is made by using the command:
`rmiregistry &`.

The server should then be started using the command: `java AloServidor`.

4.2.5 Running a Client

The client runs like any other Java program, `java AloCliente`, but it demands that the classes AloCliente, AloInterface, AloInterface_FlexRMI (generated by the `flexrmic` compiler), and Alo_Stub (generated by the `rmic` compiler) be available in the client machine. In particular the `AloCliente.class`, `AloInterface.class`, `AloInterface_FLexRMI.class` and `Alo_Stub.class` files must be in one of the directories specified in the environment variable CLASSPATH.

5 Implementation of FlexRMI

The implementation of FlexRMI is built on the available structure of the standard Java RMI. The communication protocols used by the *stubs* and *skeletons* remain unchangeable. Tasks like maintaining a socket connection with the remote server, marshaling or serializing parameters for the invoked method, and performing and blocking the remote invocation are carried out in the same way as in a standard Java RMI [3].

The core concepts in the implementation of FlexRMI are computational reflection on threads. An asynchronous remote method invocation is done dispatching, in a thread that executes in parallel with the principal thread, the computation that should be done. Identifying if the communication must be synchronous or asynchronous it just a question to distinguish the return type of the invocation made. In FlexRMI, asynchronous calls are identified by its return type, known as `Promises`[1]. So, when a call to a method is performed, it is possible to identify the type of its returned value by means of reflection of Java. Following will be discussed some FlexRMI implementation details.

5.1 The Promises Class

The class `Promises` represents a data type used as a place holder for a return value of remote asynchronous invocations. To do that, an object of type `Promises` is passed as parameter during the thread creation, which will run in parallel with the program that performed remote method invocation.

An object of the class Promises will hold the possible results of the call. This object may be inspected at any time, after the invocation is done, to see whether the return value is available or not. `Promises` objects may also store the names and types of the possible exceptions raised during execution of the remote method. In `Promises`, there is a *flag* to indicate the presence or absence of exceptions. If an exception was raised during the remote method execution, the `Promises` object will store the exception information instead of the return value of the call.

5.2 The Mediator Class

The `Mediator` class is responsible for the implementation of an object of a dynamic proxy class which is used to intercept calls and to distinguish them as synchronous or asynchronous. A dynamic proxy class is a class that implements a list of interfaces specified at execution time, such that the method invocation in an object can be captured and dispatched to another object through a standard interface.

As stated before, the calls are distinguished by the return type. When a return type `Promises` is identified, the mediator should construct a new thread and dispatch it to execution.

[1] Here are included the original methods defined to return Promises and the interfaces dynamically generated by `flexrmic`, i.e., with the methods prefixed by `async_`.

It is important to assure the creation of the mediator object in order to capture the processed calls. In our implementation, when a client asks for the remote object reference, automatically is created a mediator associated to that remote objet. To accomplish that, the `lookup` method of the `Naming` class is encapsulated in a new `lookup` method defined in the `FlexRMI` class.

5.3 Execution Scheduler

The creation of threads in an uncontrollable way could be critical in our system, affecting significantly the performance of FlexRMI. In order to avoid that, it was implemented a multi-threaded scheduler responsible to manage the process of creation, management and destruction of threads.

The scheduler executes using a list of threads. Each client has a list containing all enqueued calls. Such lists have the necessary information to execute the threads and they are periodically checked in order to guarantee the maximum number of simultaneous threads execution [2].

5.4 The flexrmic Compiler

The `flexrmic` compiler allows the client to invoke asynchronously remote methods that was designed to run synchronously, i.e., methods that do not return value of the type `Promises`.

In order to provide this facilities the compiler accesses the compiled code of remote objects and from informations thus gathered creates a new interface containing the asynchronous version of all existing methods.

To implement the `flexrmic` compiler, we use the reflection mechanism of Java. Reflection allows a Java program to be internally inspected and manipulated by itself or by another program. In this way, it was possible to find out the return type of methods, and also to exam their names and to identify those with the prefix `async_`.

The mediator function is to identify the invocation of methods present in the newly created interface. Since there is no implementation of such methods, it is necessary to dispatch their synchronous versions in others *threads*, simulating in this way the asynchronous communication.

6 Evaluation

FlexRMI allows asynchronous communication in Java without any modifications to the language or any its tools. FlexRMI is compatible with earlier Java version, allowing, inclusively, asynchronous invocations to object methods designed to be running synchronous. Additionally, our proposed interface is quite simple.

[2] This value is a system parameter. In this implementation, the standard value is 8.

On the other hand, the use of proxy implies an indirection levels in object references, which can cause degradation in the system performance. Nevertheless, this cost is compensated by the conformance of FlexRMI to the Java language and its development environment.

Since FlexRMI is dependent on Java computational reflection resources, it can not be used in environments without this property. So, its use is restricted to J2SE and J2EE version. Particularly, it can not be used by J2ME.

7 Conclusion and Further Work

Java RMI is a model of distributed computation that can be used by applications fully developed in Java. Despite it be a consolidated model, it is not adequate in many applications due to the fact that it does not allow asynchronous invocation of methods. We propose an extension mechanism, the FlexRMI, to overcome the Java RMI shortcomings.

Our implementation is fully compatible with Java RMI. Beside that, we provide conditions to allow existing synchronous services to be used in asynchronous way.

The FlexRMI package is totally operational, although some extensions should yet be implemented, to make it more efficient and functional. The two most important extensions are: the *callback* mechanism and an specific exception handler device.

Acknowledgements

This paper was developed as part of a research project sponsored by FAPEMIG (process CEX 488/2002).

References

1. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 2nd edition, 1997.
2. K. Baclawski. Java RMI Tutorial. http://www.ccs.neu.edu, 1998.
3. K. E. K. Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. In *In Proceedings of Parallel and Real Time Systems (PART'99)*, Melbourne, AUS, July 1999.
4. R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transaction on Programming Languages and Systems*, 4(7):1–1, October 1985.
5. S. Haridi, P. V. Roy, P. Brand, and C. Schulte. Programming Languages for Distributed Applications. *New Generation Computing*, 16(3):223–261, 1998.
6. J. Jaworski. *Java 2 Plataform - Unleashed.* Sams, 1st edition, 1999.
7. M. Karaorman and J. Bruno. A-RMI: Active Remote Method Invocation System for Distributed Computing Using Active Java Objects. In *In Proceedings for the Technology of Object Oriented Languages and Systems (TOOLS-26)*, Santa Barbara, California, Nevada USA, August 1998.

8. B. Liskov. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN '88*, Atlanta, Georgia USA, June 1988.
9. OMG. The Common Object Request Broker: Architecture and Specification. *Object Management Group*, December 2001.
10. R. R. Raje, J. I. William, and M. Boyles. An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. In *On-line Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation*, Las Vegas, Nevada USA, 1997.
11. D. C. Schmidt and S. Vinoski. Programming Asynchronous Method Invocations with CORBA Messaging. *SIGS C++ Report Magazine*, February 1999.
12. Sun Microsystems. Sun Microsystems Java Remote Method Invocation Specification, December 2001.
13. M. Thuan-Duc. Test Bed for Distributed Object Technologies using Java 490.45DT Project Report. citeseer.nj.ncc.com/511765.html, November 1998.
14. S. Vinoski. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), 1997.
15. J. Waldo. *The Jini Specifications*. Addison-Wesley, 2nd edition, 2001.
16. A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290, 1996.