A Distributed Object Model for the Java System

Ann Wollrath, Roger Riggs, and Jim Waldo
Sun Microsystems, Inc.

A Distributed Object Model for the Java™ System

Ann Wollrath, Roger Riggs, and Jim Waldo
JavaSoft
{ann.wollrath, roger.riggs, jim.waldo}@sun.com

Abstract

We show a distributed object model for the Java™[1] System [1,6] (hereafter referred to simply as "Java") that retains as much of the semantics of the Java object model as possible, and only includes differences where they make sense for distributed objects. The distributed object system is *simple*, in that a) distributed objects are easy to use and to implement, and b) the system itself is easily extensible and maintainable. We have designed such a model and implemented a system that supports remote method invocation (RMI) for distributed objects in Java. This system combines aspects of both the Modula-3 Network Objects system [3] and Spring's subcontract [8] and includes some novel features.

To achieve its goal of seamless integration in the language, the system exploits the use of *pickling* [14] to transmit arguments and return values and also exploits unique features of Java in order to dynamically load stub code to clients[2]. The final system will include distributed reference-counting garbage collection for distributed objects as well as lazy activation [11,16].

1 Introduction

Distributed systems require entities which reside in different address spaces, potentially on different machines, to communicate. The Java™ system (hereafter referred to simply as "Java") provides a basic communication mechanism, sockets [13]. While flexible and sufficient for general communication, the use of sockets requires the client and server using this medium to engage in some application-level protocol to encode and decode messages for exchange. Design of such protocols is cumbersome and can be error-prone.

An alternative to sockets is Remote Procedure Call (RPC) [13]. RPC systems abstract the communication interface to the level of a procedure call. Thus, instead of application programmers having to deal directly with sockets, the programmer has the illusion of calling a local procedure when, in fact, the arguments of the call are packaged up and shipped off to the remote target of the call. Such RPC systems encode arguments and return values using some type of an external data representation (e.g., XDR).

RPC, however, does not translate well into distributed object systems where communication between program-level *objects* residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require *remote method invocation* or RMI. In such systems, the programmer has the illusion of invoking a method on an object, when in fact the invocation may act on a remote object (one not resident in the caller's address space).

In order to support distributed objects in Java, we have designed a remote method invocation system that is specifically tailored to operate in the Java environment. Other RMI systems exist (such as CORBA) that can be adapted to handle Java objects, but these systems fall short of seamless integration due to their inter-operability requirement with other languages. CORBA presumes a heterogeneous, multi-language environment and thus must have a language neutral object model. In contrast, the Java language's RMI system assumes the homogeneous environment of the Java Virtual Machine, and the system can therefore follow the Java object model whenever possible.

We identify several important goals for supporting distributed objects in Java:

- support seamless remote invocation between Java objects in different virtual machines;
- integrate the distributed object model into the Java language in a natural way while retaining most of Java's object semantics;

1. Java and other Java-based names and logos are trademarks of Sun Microsystems, Inc., and refer to Sun's family of Java-branded products and services.
2. Patent pending

- make differences between the distributed object model and the local Java object model apparent;
- minimize complexity seen by the clients that use remote objects and the servers that implement them;
- preserve the safety provided by the Java runtime environment.

These goals fall under two main categories: the simplicity and naturalness of the model. It is most important that remote method invocation in Java be simple (easy to use) and natural (fit well in the language).

In addition, the RMI system should perform garbage collection of remote objects and should allow extensions such as server replication and the activation of persistent objects to service an invocation. These extensions are transparent to the client and add minimal implementation requirements on the part of the servers that use them. These additional features motivate our system-level goals. Thus, the system must support:

- several invocation capabilities
    - simple invocation (unicast)
    - invocation to multicast groups (to enable server replication)
    - extensibility to other invocation paradigms
- various reference semantics for remote objects
    - live (or non-persistent) references to remote objects
    - persistent references to and lazy activation of remote objects
- the safe Java environment provided by security managers and class loaders
- distributed garbage collection of active objects
- capability of supporting multiple transports

In this paper we will briefly describe the Java object model, then introduce our distributed object model for Java. We will also describe the system architecture and relevant system interfaces. Finally, we discuss related work and conclusions.

2 Java Object Model

Java is a strongly-typed object-oriented language with a C-style syntax. The language incorporates many ideas from languages such as Smalltalk [5], Modula-3 [10], Objective C [12] and C++ [4]. Java attempts to be simple and safe while presenting a rich set of features in the object-oriented domain.

Interfaces and Classes

One of the interesting features of Java is its separation of the notion of interface and class. Many object-oriented languages have the abstraction of "class" but provide no direct support (at the language level) for interfaces.

An *interface*, in Java, describes a set of methods for an object, but provides no implementation. A *class*, on the other hand, can describe as well as implement methods. A class may also include *fields* to hold data, but interfaces cannot. Thus, a class is the implementation vehicle in Java; an interface provides a powerful abstraction that contains no implementation detail.

Java allows subtyping of interfaces and classes by the use of *extension*. An interface may extend one or more interfaces; this capability is known as multiple-inheritance. Classes, however, are single-inheritance and may extend at most one other class.

While a class may extend at most one other class, it may *implement* any number of interfaces. A class that implements an interface provides implementations for all the methods described in that interface. If a class is defined to implement an interface, but does not provide an implementation for a particular method of that interface, it must declare that method to be *abstract*. A class containing abstract methods may not be instantiated.

An example of an arbitrary class definition in Java is as follows:

```
class Bar
      extends Foo
      implements Ping, Pong { ... }
```

where Bar is the class name, Foo is the name of the class being extended and Ping and Pong are the names of interfaces implemented by the class Bar.

Object Class Methods

All classes in Java extend the class Object, either implicitly or explicitly. The class Object has several methods which an extended class can override to have behavior specific to that class. These methods are:

- equals — tests the argument for equality with the object
- hashCode — returns a hash code for the object
- toString — returns a string representing the object
- clone — returns a clone of the object
- finalize — called to allow cleanup when the object is garbage collected

These methods are integral to the semantics of objects in Java.

Method Invocation

Method invocation in Java has the following syntax:

```
result = object.method(arg1, arg2, ...);
```

where: object is the entity which is being acted upon, method is the name of the method being called, argN is a parameter to the method, and result is the return value.

Method Parameters and Return Values

In Java, all parameters to and return values from a method are passed *by-value*. Only *references* to objects exist in Java, so object references (not objects) are passed by value. Thus, a change to an object passed to a method will be visible to the caller of the method.

The type of an object passed polymorphically does not change the type of the underlying object.

3 Distributed Object Model

In our model, a *remote object* is one whose methods can be accessed from another address space, potentially on a different machine. An object of this type is described by a *remote interface*, which is an interface (in Java) that declares the methods of a remote object. Remote method invocation (or RMI) is the action of invoking a method (of a remote interface) on a remote object. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

Clients of remote objects program to remote interfaces, not to the implementation classes of those interfaces. Since the failure modes of accessing remote objects are inherently different than the failure semantics of local objects, clients must deal with an additional exception that can occur during any remote method invocation.

What follows is a brief comparison of the distributed object model and the Java object model. The similarities between the models are:

- a reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote);
- a remote object can be cast to any of the 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.

There are several basic differences between the distributed object model and the Java object model:

- 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;
- parameter passing semantics are slightly different in calls to remote objects;
- semantics of Object methods are defined to make sense for remote objects.

Remote Interfaces

In order to implement a remote object, one must first define a remote interface for that object. A remote interface must extend (either directly or indirectly) a distinguished interface called java.rmi.Remote. This interface is completely abstract and has no methods.

```
interface Remote {}
```

For example, the following code fragment defines a remote interface for a bank account that contains methods that deposit to the account, withdraw from the account, and get the account balance:

```
import java.rmi.*;

public interface BankAccount
        extends Remote
{
     public void deposit(float amount)
           throws RemoteException;
     public void withdraw(float amount)
           throws OverdrawnException,
                  RemoteException;
     public float balance()
           throws RemoteException;
}
```

As shown above, each method declared in an interface for a remote object must include java.rmi.RemoteException in its throws clause. If RemoteException is thrown during a remote call, then some communication failure happened during the call. Remote objects have very different failure semantics than local objects. These failures cannot be hidden from the programmer since they cannot be masked by the underlying system [15]. Therefore, we choose to expose the additional exception RemoteException in all remote method calls, so that programmers can handle this failure appropriately.

Remote Implementations

There are two ways to implement a remote interface (such as BankAccount). The simplest implementation route is for the implementation class, e.g., BankAcctImpl, to *extend* the class RemoteServer. We call this first scheme *remote implementation reuse*. Figure 1 below is an illustration of the interface and class hierarchies for remote interfaces and implementations in this scheme.

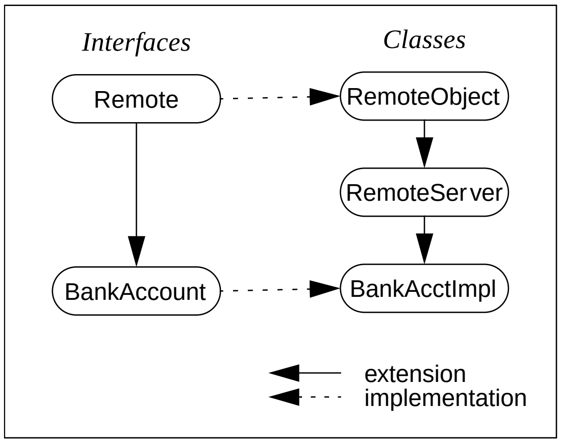

Figure 1. Reusing a remote implementation

The default constructor for RemoteServer takes care of making an implementation object remotely accessible to clients by *exporting* the remote object implementation to the RMI runtime. The class RemoteObject overrides methods inherited from Object to have semantics that make sense for remote objects. We will discuss what the appropriate semantics for these methods are in the section on “Object Method Semantics”.

In the second implementation scheme, called *local implementation reuse*, the implementation class for a remote object does not extend RemoteServer but may extend any other local implementation class as appropriate. However, the implementation must explicitly export the object to make it remotely accessible.

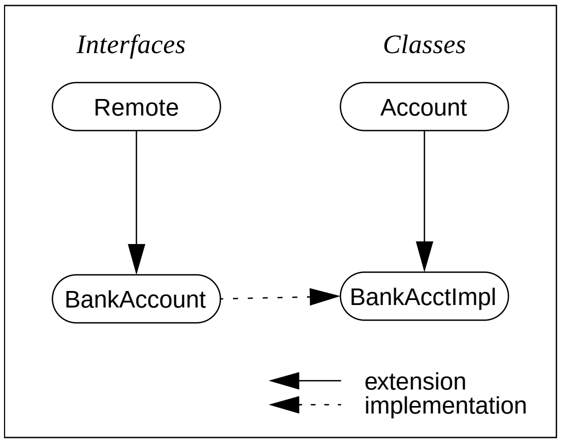

Figure 2. Reusing a local implementation class

The “local implementation reuse” scheme (shown in Figure 2), while allowing the class to reuse existing implementation code, does require that the class deal with the details of making instances of that class remotely accessible (by exporting the object to the RMI runtime). Such exporting is already taken care of in the RemoteServer constructor used in the first scheme.

Implementations using the second scheme must also be responsible for their own Java Object semantics and therefore must redefine methods inherited from the class Object appropriately. These object methods are already taken care of in the implementation of RemoteObject, used in the other scheme.

We deem the “remote implementation reuse” scheme more seamlessly integrated into the Java object model as well as requiring less implementation detail; so we will explain that implementation scheme in depth here. The other scheme, local implementation reuse, is included for implementation flexibility if such is required by the programmer.

Thus, BankAcctImpl, an implementation class of the remote interface BankAccount can be defined by extending RemoteServer as follows and would implement all the methods of BankAccount:

```
package myPackage;

import java.rmi.RemoteException;
import java.rmi.server.RemoteServer;

public class BankAcctImpl
          extends RemoteServer
          implements BankAccount
{
     public void deposit(float amount)
          throws RemoteException {...};
     public void withdraw(float amount)
          throws OverdrawnException,
                 RemoteException {...};
     public float balance()
          throws RemoteException {...};
}
```

A few additional notes about implementing remote interfaces are:

- An implementation class may implement any number of remote interfaces.
- An implementation class may extend any other implementation class of a remote interface.
- *Only* those methods that appear in a remote interface (one that extends Remote either directly or indirectly) can be accessed remotely; thus non-remote methods in an implementation class can only be accessed locally.

The server implementation scheme fits very well into the Java object model and the Java language.

Remote Reference Types

In the distributed object model, clients interact with stub (surrogate) objects that have *exactly* the same set of remote interfaces defined by the remote object's class; the stub class does not include the non-remote portions of the class hierarchy that constitutes the object's type graph. This is because the stub class is generated from the most refined implementation class that implements one or more remote interfaces. For example, if C extends B and B extends A, but only B implements a remote interface, then a stub is generated from B, not C.

Because the stub implements the same set of remote interfaces as the remote object's class, the stub has, from the point of view of the Java system, the same type as the remote portions of the server object's type graph. A client, therefore, can make use of the built-in Java operations to check a remote object's type and to cast from one remote interface to another, e.g.:

```
Remote obj = ...;// lookup object
if (obj instanceof BankAccount) {
     BankAccount acct = (BankAccount)obj;
     //...
}
```

The system employs a mechanism called *dynamic stub loading* to make the correct stub for the remote object available to the client (this technique is fully described in the section "System Architecture").

Remote Method Invocation

For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is obtained in the usual manner: as a return value in a method call or as a parameter passed to a method. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts.

Invoking a method on a remote object has the same syntax as invoking a method on any Java object. For example, here's how the bank account could be accessed (without exception handling):

```
BankAccount acct = ...;// lookup account
float balance;
acct.deposit(243.50);
acct.withdraw(100.00);
balance = acct.balance();
```

Since remote methods include RemoteException in their signature, the caller must be prepared to handle those exceptions in addition to other application specific exceptions. So, for each of the calls above (deposit, withdraw, and balance), the code needs to catch RemoteException (and the withdraw call would need to also catch OverdrawnException).

If RemoteException is thrown during a remote call, then some communication failure happened during the call. The client has little to no information on the outcome of the call—whether a failure happened before, during, or after the call completed. Thus, remote interfaces should be designed with these failure semantics in mind [9,15]. The semantics of a remote method may need to be idempotent whereas calls within the local address space likely do not have to be. Note that the above bank account interface does not support idempotent operations, so if an operation fails, the client needs to perform some type of recovery to determine the true state of the bank account (using transactions would solve this problem).

In most cases, a method invoked on a remote object is indirected through the remote object's stub to which the caller has a reference. In a method invocation to a remote object which actually *resides* in the same virtual machine as the caller, the call *may* be a local invocation and not a call via the stub for the remote object. If the caller has an actual reference to the remote object implementation, the method call is local and is not forwarded via a stub. However, a caller may receive, from a remote object in a different virtual machine, a remote reference to the object whose implementation is in the same virtual machine. In this case, the client (the caller) has a reference to a stub for the remote object; thus, a method call on this reference would be indirected through the stub.

Object Method Semantics

The default implementations for the methods of class Object (equals, hashCode, toString, clone, and finalize) are not appropriate for remote objects. The class RemoteObject provides implementations for these methods that have semantics more appropriate for remote objects.

In order for a remote object to be used as a key in a hash table, the methods equals and hashCode need to be overridden in a remote object implementation. The semantics of equals for a remote object must be defined such that remote objects have *reference equality*. Thus, given any two remote references to the same underlying object, those objects will be equal. No stronger equality, such as "content" equality, may be defined for remote objects, since determining the equality of contents would require a remote call. Remember that in a remote call, a RemoteException may be raised, and the method equals has no such exception in its throws clause. Due to the different failure semantics between local and remote calls, we chose to implement only reference equality for remote objects.

The hashCode method will return the same value for remote references that refer to the same underlying object.

The toString method is defined to return a string which represents the reference of the object. In the current implementation that supports unicast method invocation, the contents of this string includes transport specific information about the object (e.g., host name and port number) and an object identifier.

Objects are only cloneable using the Java language's default mechanism if they support the java.lang.Cloneable interface. Remote objects do not implement this interface, but do implement the clone method so that if subclasses need to implement Cloneable, the remote classes will function correctly.

Cloning a reference to a remote object is a local operation and cannot be used by clients to create a new remote object.

For RemoteServer objects, clone is implemented to make a new remote object distinct from the original. Cloning a remote object is only available in the server process where the remote object exists. If a remote object does not extend RemoteServer, it must implement its own version of clone and be able to export a cloned object.

The clone method for a remote object is defined to return a reference to the remote object. This operation does not copy any contents of the remote object, it simply returns a reference (since determining contents would require a remote call, and clone does not have RemoteException in its throws clause which would be raised in the event of a remote call failure).

The finalize method is used in specific circumstances depending on the type of remote object (for example, if a remote object is one that can be activated, some cleanup may be necessary).

There are several other methods defined in the class Object. These methods, however, are declared as final, which means that they cannot be overridden in an extended class. The methods are: getClass, notify, notifyAll, and wait.

The default implementation for getClass is appropriate for all Java objects, local or remote. The method needs no special implementation for remote objects. When used on a remote object, the getClass method reports the exact type of the generated stub object. Note that this type reflects only the remote interfaces implemented by the object, not its local interfaces.

The wait/notify methods of Object deal with waiting and notification in the context of Java's threading model. While use of these methods for remote objects does not break the Java threading model, these methods do not have the same semantics as they do for local Java objects. Use of these methods would only operate on the client's local reference to the remote object, not the actual object at the remote site. Since these methods are final, they cannot be extended to have behavior specific to remote objects.

Due to the differing failure modes of local and remote objects, distributed wait and notification requires a more sophisticated protocol between the entities involved (so that, for example, a client crash does not cause a remote object to be locked forever), and as such, cannot be easily fitted into the local threading model in Java. Hence, a client can use notify and wait methods on a remote reference, but that client must be aware that such actions will not involve the actual remote object, only the local proxy (stub) for the remote object.

Parameter Passing in Remote Invocation

A parameter of any Java type can be passed in a remote call. These types include both Java primitive types and Java objects (both remote and non-remote).

The parameter passing semantics for remote calls are the same as the Java semantics *except*:

- non-remote objects contained in a parameter of a remote call are passed by *copy*; and,
- non-remote objects returned as the result of a remote call are also passed by *copy.*

That is, when a non-remote object is passed in a remote call, the content of the non-remote object is copied before invoking the call on the remote object. Thus, there is no relationship between the non-remote object the client holds and the one it sends to a remote server in a call. For example, let's suppose that the remote object bank has a method to obtain the bank account given a name and social security number; the account information info is not a remote object but a local Java object:

```
Bank bank = ...;
String ssn = "999-999-9999";
AccountInfo info =
     new AccountInfo("Robin Smith", ssn);
BankAccount acct = bank.getAccount(info);
```

```
info.setName("Robyn Smith");
```

The contents of the object info is copied before invoking the remote call on the bank. A client can make changes to info without effecting the server's copy and vice versa.

Locating Remote Objects

A simple bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based interface java.rmi.Naming.

For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is usually obtained as a return value in a method call. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts. The Naming interface provides Uniform Resource Locator (URL) based methods to lookup, bind, rebind, unbind and list the name and object pairings maintained on a particular host and port.

Here's an example of how to bind and lookup remote objects:

```
BankAccount acct = new BankAcctImpl();
URL url = new URL(“rmi://zaphod/account”);
// bind url to remote object
java.rmi.Naming.bind(url, acct);
// ...
// lookup account
acct = java.rmi.Naming.lookup(url);
```

In the current implementation, a “naming” registry contains a non-persistent database of name-object bindings. This database does not survive system crashes.

4 System Architecture

We have designed our RMI system in order to support the distributed object model discussed above. The system consists of three basic layers: the *stub/skeleton layer*, *remote reference layer*, and *transport*. A specific interface and protocol defines the boundary at each layer. Thus, each layer is independent of the next and can be replaced by an alternate implementation without effecting the other layers in the system. For example, the current transport implementation is TCP-based (using Java sockets), but a transport based on UDP could be used interchangeably.

To accomplish transparent transmission of objects from one address space to another, the technique of *pickling* [14] (designed specifically for Java) is used.

Another technique, that we call *dynamic stub loading*, is used to support client-side stubs which implement the same set of remote interfaces as a remote object itself. Since a stub of the exact type is available to the client of a remote object, a client can use Java’s built-in operators for casting and typechecking remote interfaces.

Architectural Overview

The three layers of the RMI system consist of the following:

- stub/skeletons — client-side stubs (proxies) and server-side skeletons (dispatchers)
- remote reference layer — invocation behavior and reference semantics (e.g., unicast, multicast)
- transport — connection set up and management and remote object tracking

The application layer sits on top of the RMI system.

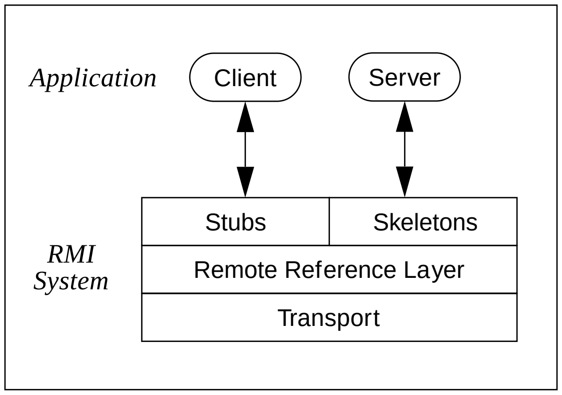

Figure 3. System Architecture

Figure 3 is an illustration of the layers of the RMI system. A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server. The rest of this section summarizes the functionality at each layer in the system.

A client invoking a method on a remote server object actually makes use of a *stub* or proxy for the remote object as a conduit to the remote object. A client-held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote object and forwards invocation requests to that server object via the remote reference layer.

The *remote reference layer* is responsible for carrying out the semantics of the *type* of invocation. For example this layer is responsible for handling unicast or multicast invocation to a server. Each remote object implementation chooses its own invocation semantics—whether communication to the server is unicast, or the server is part of a multicast group (to accomplish server replication).

Also handled by the remote reference layer are the reference semantics for the server. For example, the remote reference layer handles live and/or persistent references to remote objects. Persistent object references are required in order to activate objects to support long-running servers.

The *transport* is responsible for connection set-up with remote locations and connection management, and also keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport's local address space.

In order to dispatch to a remote object, the server's transport forwards the remote call up to the remote reference layer (specific to the server). The remote reference layer handles any server-side behavior that needs to be done before handing off the request to the server-side skeleton. The skeleton for a remote object makes an up-call to the remote object implementation which carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer and transport on the server side, and then up through the transport, remote reference layer and stub on the client side.

Stub/Skeleton Layer

The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. This layer does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of *marshal streams*. Marshal streams employ a mechanism called *pickling* which enables Java objects to be transmitted between address spaces. Objects transmitted using the pickling system are passed by copy to the remote address space.

A *stub* for a remote object is the client-side proxy for the remote object. Such a stub implements all the interfaces that are supported by the remote object implementation. A client-side stub is responsible for:

- initiating a call to the remote object (by calling the remote reference layer)
- marshaling arguments to a marshal stream (obtained from the remote reference layer)
- informing the remote reference layer that the call should be invoked
- unmarshaling the return value from a marshal stream
- informing the remote reference layer that the call is complete

A *skeleton* for a remote object is a server-side entity that contains a method which dispatches calls to the actual remote object implementation. The skeleton is responsible for:

- unmarshaling arguments from the marshal stream
- making the up-call to the actual remote object implementation
- marshaling the return value of the call onto the marshal stream

Remote Reference Layer

The remote reference layer deals with the lower level transport interface. This layer is also responsible for carrying out a specific invocation protocol which is independent of the client stubs and server skeletons.

Each remote object implementation chooses its own invocation protocol that operates on its behalf. Such an invocation protocol is fixed for the life of the object. Various invocation protocols can be carried out at this layer, for example:

- unicast invocation
- multicast invocation
- support for a specific replication strategy
- support for a persistent reference to the remote object (enabling activation of the remote object)
- reconnection strategies (if remote object becomes inaccessible)

These invocation protocols are not mutually exclusive, but may be combined. For example, a remote object may require both persistent reference semantics and replication. Both of these protocols would be carried out in the remote reference layer.

The invocation protocol is divided into two cooperating components: the client-side and the server-side components. The client-side component contains information specific to the remote server (or servers, if invocation is to a multicast group) and communicates via the transport to the server-side component. During each method invocation, the client and server-side components are given a chance to intervene in order to accomplish the specific invocation and reference semantics. For example, if a remote object is part of a multicast group, the client-side component can forward the invocation to the multicast group rather than just a single remote object.

In a corresponding manner, the server-side component is given a chance to intervene before delivering a remote method invocation to the skeleton. This component, for example, could handle ensuring atomic multicast delivery by communicating with other replicas in the multicast group.

The remote reference layer transmits data to the transport layer via the abstraction of a stream-oriented *connection*. The transport takes care of the implementation details of connections. Although connections present a streams-based interface, a connectionless transport may actually be implemented beneath the abstraction.

Transport

In general, the transport of the RMI system is responsible for:

- setting up connections to remote address spaces
- managing connections
- monitoring connection liveness
- listening for incoming calls
- maintaining a table of remote objects that reside in the local address space
- setting up a connection for an incoming call
- locating the dispatcher for the target of the remote call and passing the connection to this dispatcher

The concrete representation of a remote object reference consists of an endpoint and an object identifier. We call this representation a *live reference*. Thus, given a live reference for a remote object, a transport can use the endpoint to set up a communication channel to the address space in which the remote object resides. On the server side, the transport uses the object identifier to look up the target of the remote call.

The transport for the RMI system consists of four basic abstractions (based somewhat on the transport of the Modula-3 network object system):

- Endpoint — An endpoint denotes an address space. In the implementation, an endpoint can be mapped to its transport. That is, given an endpoint, a specific transport instance can be obtained.
- Transport — The transport abstraction manages channels. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces, the local address space and a remote address space. Given an endpoint to a remote address space, a transport sets up a channel to that address space. The transport abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system.
- Channel — Abstractly, a channel is the conduit between two address spaces. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel.
- Connection —A connection is the abstraction for transferring data (performing input/output).

A transport defines what the concrete representation of an endpoint is, so multiple transport implementations may exist. The design and implementation also allow multiple transports per address space (so both TCP and UDP can be supported in the same address space). Thus, client and server transports can negotiate to find a common transport between them.

Garbage Collection

In a distributed system, just as in the local system, it is desirable to automatically delete those remote objects that are no longer referenced by any client. This frees the programmer from needing to keep track of a remote object's clients so that the remote object can terminate appropriately. RMI uses a reference counting garbage collection algorithm similar to the one used for Modula-3 Network Objects [2].

To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each Java virtual machine. When a live reference enters a Java virtual machine its reference count is incremented. The first reference to an object sends a "referenced" message to the server for the object. As live references are found to be unreferenced in the local virtual machine, their finalization decrements the count. When the last reference has been discarded an unreferenced message is sent to the server. Many subtleties exist in the protocol, most related to maintaining the ordering of referenced and unreferenced messages to insure the object is not prematurely collected.

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine's garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine's garbage collector in the usual ways by holding normal or weak references to objects. As in the normal object life-cycle, finalize will be called after the garbage collector determines that no more references to the object exist.

As long as a local or remote reference to a remote object exists, the remote object cannot be garbage collected and it may be passed in remote calls or returned to clients. Passing a remote object adds the client or server to which it was passed to the remote object's referenced set. A remote object needing unreferenced notification must implement the java.rmi.server.Unreferenced interface. When those references no longer exist, unreferenced will be invoked. unreferenced is called when the set of references is found to be empty so it may be called more than once. Remote objects are only collected when no more references, either local or remote, still exist.

Note that if there exists a network partition between a client and remote server object, it is possible that premature collection of the remote object will occur (since the transport may think that the client crashed). Because of the possibility of premature collection, remote references cannot guarantee referential integrity; in other words, it is always possible that a remote reference may in fact not refer to an existing object. An attempt to use such a reference will generate a RemoteException which must be handled by the application.

Dynamic Stub Loading

In remote procedure call systems, client-side stub code must be generated and linked into a client before a remote procedure call can be done. This code may be either statically linked into the client or linked in at run-time via dynamic linking with libraries available locally or over a network file system. In either the case of static or dynamic linking, the specific code to handle an RPC must be available to the client machine in compiled form.

This approach to code linking is static in that the stub code must be compiled and directly available to the client in binary-compatible form at compile time and at run time. Also with these systems, the stub code that the client uses is determined and fixed at compile time. Because of the static nature of the stub code available to clients in such systems, the code may not be the actual stub code that the client needs at run time, but perhaps the closest matched code that can be determined at compile time. For example in an RMI system, perhaps only a supertype (less specific form) of a more specific stub is available to the client at run-time. This code mismatch can lead to run-time errors if the client in fact needs a subtype (more specific form) of the stub that has been linked in at compile-time.

Our approach solves this code mismatch by loading the exact stub code (in Java's architecture neutral bytecode format) at run-time to handle method invocations on a remote object. This mechanism, called *dynamic stub loading*, exploits the Java mechanism for downloading code.

Dynamic stub loading is used only when code for a needed stub is not already available. The argument and return types specified in the remote interfaces are made available using the same mechanism. Loading arbitrary classes into clients or servers presents a potential security problem; this problem is addressed by requiring that a security manager check any classes downloaded for RMI.

In this scheme, client-side stub code is generated from the remote object implementation class, and therefore supports the same set of remote interfaces as supported by the remote implementation. Such stub code resides on the server's host (or perhaps another location), and can be downloaded to the client on demand (if the correct stub code is not already available to the client). Stub code for a remote implementation could be generated on-the-fly at the remote site and shipped to the client or could be generated on the client-side from the list of remote interfaces supported by the remote implementation.

Dynamic stub loading employs three mechanisms: a specialized Java class loader, a security manager, and the pickling system. When a remote object reference is passed as a parameter or as the result of a method call, the marshal stream that transmits the reference includes information indicating where the stub class for the remote object can be loaded from, if its URL is known.

A marshal stream is implemented by an underlying pickle stream. Pickle streams provide an opportunity to embed information for each class and object that is transmitted. When transmitting class information for a remote object being marshaled, a marshal stream embeds a URL that specifies where the stub code resides. Thus, when a reference to a remote object is unmarshaled at the destination site, the marshal stream can locate and load the stub code (via the specialized class loader, checked by the security manager) so that the correct stub is available to the client.

Security

The security of a process using RMI is protected by existing Java mechanisms of the security manager and class loader. The security manager regulates access to sensitive functions, and the class loader makes sure that loaded classes are subject to the security manager and adhere to the standard Java safety guarantees.

The JDK (Java Developer Kit) 1.0 security manager does not regulate resource consumption, so the current RMI system has no mechanisms available to prevent classes loaded from abusing resources. As new security manager mechanisms are developed to control resource use, RMI will use them.

The Applet Environment

In the applet environment, the AppletSecurityManager and AppletClassLoader are used exclusively. RMI uses only the established security manager and class loader. In this environment remote object stubs, parameter classes and return object classes can be loaded only from the applet host or its designated code base hosts. This requires that applet developers install the appropriate classes on the applet host.

The Server Environment

In the server environment, where a Java process is being used to serve RMI requests, the server may need to use a security manager to isolate itself from stub misbehavior. The server functions will usually be implemented by classes loaded from the local system and therefore not subject to the restrictions of the security manager. If the remote object interfaces allow objects, either local or remote, to be passed to the server, then those object classes must be accessible to the server. Usually those classes will be built-in classes or will be defined by the server. As long the classes are available locally there is no need for a specialized security manager or stub loader. To support this case, if no security manager is specified, stub loading from network sources is disabled.

When a server is passed a remote object for which it has no corresponding stub code, it may also be passed the location from which the classes for that remote object may be loaded. Two properties control if and from where the stub class can be loaded.

java.rmi.server.ClientClassDisable controls whether the URL's supplied by clients are used; if set to true, URL's from clients are ignored and stub classes are loaded using the stub class base.

java.rmi.server.StubClassBase defines the URL from which stub classes will be loaded. This is the URL that is passed along with remote object references so clients will know the location from which to load stub classes.

The StubClassLoader is a specialized class loader used by the RMI runtime to load classes. When loading any class, the runtime first attempts to use this class loader. If it succeeds those classes will be subject to the current security manager and any classes the stub needs will be loaded and then regulated by that security manager. If the security manager disallows creating the class loader, the class (including stub classes) will be loaded using the default Class.forName mechanism. Thus, a server may define its own policies via the security manager and stub loader and the RMI system will operate within them.

5 Related Work

The Common Object Request Broker Architecture (CORBA) [11] is designed to support remote method invocation between heterogeneous languages and systems. In CORBA, distributed objects are described in an interface definition language (IDL). IDL presents its own object model, and interfaces defined in IDL must be mapped into a target language and object model.

Because of IDL's language neutrality, the semantics of its object model does not match the object model semantics of any implementation language. This mismatch inhibits seamless integration of the CORBA distributed object model into any specific target language. Hence, programmers must deal with two very different object models when writing distributed programs: the local object model of the language, and the distributed object model mapped from IDL.

Our system differs from CORBA in two essential ways: it language-dependence and its ability to load stub code dynamically. Since our system is language-dependent, we can integrate the distributed object model more closely with the target language, Java. Also, systems that are CORBA compliant are unable to exploit the use of dynamic stub loading, since CORBA generally assumes that stub code is linked in at compile time.

Our approach is more akin to the Modula-3 (M3) network object system [3]. The Modula-3 system supports remote method invocation on objects in a language-dependent fashion (i.e., the system does not support interoperability with other languages). A second similarity is that the M3 system transmits objects via pickling. Our RMI system is similar in those respects (it depends on the architecture neutrality of Java bytecodes); however, our system is less static in its determination of stub code. The M3 network object system uses the closest matching stub code (called the most-specific surrogate) available at compile-time, rather than our approach in which the exact matching stub code is determined at run-time and downloaded over the network if such code is unavailable on the client.

The implementation of our system is similar to the M3 system in another respect: that is, the inclusion of a distinct abstraction for the transport. While the network object system has a similar notion of a transport abstraction, it does not include a separate remote reference layer to handle varying types of invocation semantics. Because of this limitation, this type of functionality is not easily layered on the network object system without adding some burden to the programmer.

Spring [7] is an object oriented operating system designed as a successor to UNIX. Spring has the notion of a subcontract [8] which has similar functionality to the remote reference layer in the RMI system. Our system differs in that the remote reference layer has a narrower interface that is more tailored to handling remote method invocation semantics. Subcontract is also very intimate with its doors-based transport, and as such does not support alternate transport implementations as readily as our approach.

Like CORBA, Spring uses an interface definition language to describe remote objects. Spring uses marshaling code generated from IDL descriptions of objects, whereas our system pickles exact representations of objects at run-time.

6 Future Work

The current system supports unicast remote method invocation to remote objects in Java. The system also implements pickling, dynamic stub loading, and garbage collection. We have fully designed and partially implemented activation for distributed objects in this framework. This effort is on-going. Also included will be the capability for server replication in this paradigm.

7 Conclusions

Our RMI system design leverages the two basic assumptions of platform homogeneity and language-dependence. We can assume homogeneity due to the architecture neutrality that the Java virtual machine provides. Since we are able to focus on language-dependent distributed objects, the resulting system presents a simple model that fits well into the Java framework, is highly flexible, and is accessible on a wide variety of machines.

Availability

The Java RMI system will be released with JDK 1.1. Early access versions of this system can be obtained from the http://java.sun.com web site.

Acknowledgments

We would like to thank Eduardo Pelegri-Llopart and Peter Kessler for useful discussions during the development of this system.

Authors

Ann Wollrath (ann.wollrath@sun.com) is a Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, she worked in the Parallel Computing Group at the MITRE Corporation investigating optimistic execution schemes for parallelizing sequential object-oriented programs.

Roger Riggs (roger.riggs@sun.com) is a staff engineer at Sun Microsystems Laboratories, East Coast Division, working on large scale distribution and reliable distributed systems. Prior to joining the group, he worked on client-server text retrieval and CORBA based information retrieval, and scabability issues on the Web.

Jim Waldo (jim.waldo@sun.com) is a Senior Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked in distributed systems and object-oriented software development at Apollo Computer (later Hewlett Packard), where he was one of the original designers of what has become the Common Object Request Broker Architecture (CORBA).

References

[1] Arnold, Ken, and James Gosling, *The Java™ Programming Language*. Addison-Wesley (1996).

[2] Birrell, Andrew, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber, *Distributed Garbage Collection for Network Objects*. Digital Equipment Corporation Systems Research Center Technical Report 116 (1993).

[3] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, *Network Objects*. Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).

[4] Ellis, Margaret A., and Bjarne Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley (1990).

[5] Goldberg, Adele, and David Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley (1983).

[6] Gosling, James, Bill Joy, and Guy Steele, *The Java™ Language Specification*. Addison-Wesley (1996).

[7] Hamilton, G., and P. Kougiouris, “The Spring Nucleus: A Microkernel for Objects.” *USENIX Summer Conference* (July 1993).

[8] Hamilton, Graham, Michael L. Powell, and James G. Mitchell, *Subcontract: A Flexible Base for Distributed Programming*. Sun Microsystems Laboratories Technical Report, SMLI TR-93-13 (April 1993).

[9] Mullender, Sape (ed.), *Distributed Systems* (second edition). Addison-Wesley (1993).

[10] Nelson, Greg (ed.), *Systems Programming with Modula-3*. Prentice Hall (1991).

[11] The Object Management Group, *Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1 (1991).

[12] Pinson, Lewis J. and Richard S. Wiener, *Objective C: Object-Oriented Programming Techniques*. Addison-Wesley (1991).

[13] Rago, Steven A., *UNIX System V Network Programming*. Addison-Wesley (1993).

[14] Riggs, Roger, Jim Waldo, Ann Wollrath, and Krishna Bharat, “Pickling State in the Java™ System.” The *2nd USENIX Conference on Object-Oriented Technologies*. (1996).

[15] Waldo, Jim, Geoff Wyant, Ann Wollrath, and Sam Kendall, *A Note on Distributed Computing*. Sun Microsystems Laboratories Technical Report, SMLI TR-94-29 (November 1994).

[16] Wollrath, Ann, Geoff Wyant, and Jim Waldo, “Simple Activation for Distributed Objects.” *1st USENIX Conference on Object-Oriented Technologies*. Monterey, CA (June 1995), pp. 1-11.