The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

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.

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

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;

- 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: