Jim Waldo
Sun Microsystems
jim.waldo@sun.com

Remote procedure calls and Java Remote Method Invocation

REMOTE PROCEDURE CALL SYSTEMS have been around since Andrew Birrell and Greg Nelson first proposed them in 1984.[1] During the intervening 15 years, numerous evolutionary improvements have occurred in the basic RPC system, leading to improved systems—such as NCS[2]—that offer programmers more functionality or greater simplicity. The Common Object Request Broker Architecture from the Object Management Group[3] and Microsoft's Distributed Common Object Model[4] are this evolutionary process's latest outgrowths.

With the introduction of Java Developer's Kit release 1.1, a third alternative for creating distributed applications has emerged. The Java Remote Method Invocation system has many of the same features of other RPC systems, letting an object running in one Java virtual machine make a method call on an object running in another, perhaps on a different physical machine.

On the surface, the RMI system is just another RPC mechanism, much like Corba and DCOM. But on closer look, RMI represents a very different evolutionary progression, one that results in a system that differs not just in detail but in the very set of assumptions made about the distributed systems in which it operates. These differences lead to differences in the programming model, capabilities, and way the mechanisms interact with the code that implements and built the distributed systems.

RPC system assumptions

To understand the differences between RMI and more standard RPC mechanisms such as Corba and DCOM, let's examine the different systems' underlying assumptions concerning their operating environment. These assumptions structure the thinking of the system's designers and the resulting system's functionality.

Corba and DCOM (and earlier systems in the RPC evolutionary line) were built on assumptions of heterogeneity. These mechanisms assume that the distributed system contains machines that might be different, running different operating systems. The machines' instruction sets might differ. Indeed, data representations might differ from one machine to another.

This heterogeneity is the central problem that systems such as Corba were designed to solve. As their solution, these systems introduced proxies on each machine that could process information passing from one member of the distributed system to another. The processing converted the information from the format known by one member into a format known by the other.

The proxy on the client (calling) side became known as a *stub*, while the proxy on the server (receiving) side became known as the *skeleton*. Stubs are compiled into the calling code and convert any call into some machine-neutral data representation (a process called *marshaling*) that gets transmitted to the receiving-machine skeleton. This skeleton will translate the transmitted information into the appropriate data types for that machine (a process known as *unmarshaling*), will identify the code that needs to be called with that information, and will make the call. Skeletons also marshal any return values, transmitting them back to the stub that made the call, which will unmarshal those values and return them to the calling code.

Stub and skeleton production happens automatically, based on language and machine-neutral interface-definition language descriptions of the calls that can be made. An IDL definition describes the procedures or methods that can be called over the network and the information that passes to and from those procedures or methods. From such a description, the programmer can invoke an IDL compiler that will produce stub and skeleton source code for marshaling, transmitting, and unmarshaling this data. This code can then be compiled for the target machine and linked into the appropriate application code.

Having a machine-neutral IDL also lets such systems deal with heterogeneous languages. Different IDL compilers can translate the remote interfaces defined through the IDL into stubs and skeletons for different implementation languages. Because the exchanged data's on-the-wire format is the same no matter what language is used, stubs generated by one compiler can send procedure or method calls (along with parameters) to a skeleton generated by a compiler for a different target language.

This approach has considerably simplified the building of

distributed applications. However, it has limitations. The language-neutral nature of these systems limits the kinds of data that can travel between processes to the basic data types that can be represented in all the target languages, to references to remote objects, and to structures made up of basic data types and references to remote objects. Although the complexity of network data representations are hidden from the programmer, the programmer must face the complexity of mapping from the IDL to the implementation-language data types. Finally, the life-cycle management of data sent, either as a parameter or a return, requires complex conventions or explicit reference counting, both of which are subject to programmer error that can cause memory leaks or referential integrity loss.

Perhaps this approach's greatest limitation, however, is the static nature of the information that can pass over the network. Systems built this way depend on the stub and skeleton matching, allowing the receiving process to interpret the sent information. This in turn requires that the receiving process know exactly what the sending process places on the wire. In object-oriented terms, no polymorphism is allowed—the transmitted object's type (or its reference type) cannot be a subtype of the type expected by the skeleton. If a process passes an instance or a reference to such a subtype, the system converts it to a reference of the exact type that the skeleton expects.

RMI system assumptions

The Java RMI system is built on an entirely different set of assumptions. Heterogeneity is not the major problem. Indeed, it is not a problem at all, because RMI assumes that the client and the server are both Java classes running in a Java virtual machine, which makes the network a homogeneous collection of (virtual) machines.

The RMI system takes homogeneity one step further and assumes that all objects constituting the distributed system are written in Java. RMI's designers made this single-language assumption to simplify the overall system, placing RMI in the evolutionary line of language-centric systems such as Oberon and the Modula-3 Network Object system. With this single-implementation language assumption, the RMI system does not need a language-neutral IDL. RMI simply uses the Java `interface` construction to declare remotely accessible interfaces. A remote interface in RMI is one that extends the marker interface `java.rmi.Remote`.

Finally, RMI's Java-centric design lets the system capitalize on the Java environment's dynamic nature, letting code load any time during execution. Rather than requiring, as do traditional RPC systems, that all code needed for communication between processes be available at some time prior to that communication, RMI makes aggressive use of dynamic code loading, from stubs representing remote objects to real objects that can pass from one part of a distributed system to another.

The ability to download code plays its most prominent role in connection with the ability to pass full objects into and out of an RMI call. Because of RMI's single-language assumption, the system allows almost any Java object to pass as a parameter or return value in a remote call. Remote objects pass by reference, in effect, by passing a copy of the object's stub code. Nonremote objects are passed by value, creating a copy of the object in the destination..

The objects that pass are real objects, not just the data that makes up an object's state. This distinction becomes important if a subtype of a declared type passes from one member of the distributed computation to another. Passing a subtype object could result in receiving an unknown object. Because a subtype can change the behavior of known methods in an object, simply treating the object as an instance of a known type might change the results of making a method call on that object.

RMI's Java-centric design lets the system capitalize on the Java environment's dynamic nature, letting code load any time during execution.

To avoid such a change, RMI uses a variant of the Java Object Serialization package to marshal and reconstruct objects. This package will annotate any object with enough information to identify the object's exact type and its implementation code. When an object of a previously unknown type is received as the result of an RMI call, the system fetches the code for that object, verifies it, and dynamically loads it into the receiving process.

Distributed computations therefore can use all the standard object-oriented design patterns that rely on polymorphism, because the object's behavior moves when the object moves. The receiving process must simply define the set of methods that will be called on an object that passes to it; how the object implements those methods can vary in ways that the receiving process need not know. Although we generally discuss passing objects in conjunction with nonremote objects that pass by value, it also affects the way the RMI system can pass references to remote objects. These references are themselves stub objects, generated by the RMI compiler. However, unlike standard RPC IDL compilers, these stubs are generated on the implementation class of the object to which the stub refers. Rather than reflecting only the declared remote interface, these stub objects support all the remote methods that the remote object's implementation supports.

Stubs for remote objects therefore also load at runtime when needed, and the stub will reflect the exact (remote) type of the object for which the stub is a proxy. This changes the basic notion of ownership and responsibility in the distributed application. In systems such as Corba, the stub code is the responsibility of the calling client and can be linked ahead of time into that client. In the RMI system, the stub for a remote object

originates with the object and can be different for any two objects with the same apparent type. The system locates and loads these stubs at runtime, when the system determines what the exact stub type is.

Such an association makes the stub an extension of the remote object in the client's address space, rather than something that is built into the client as a way of contacting the remote object. This approach allows programmers using the system to build a variety of "smart" proxies. Such smart proxies can cache certain values in the stub, avoiding the need to make remote calls in certain cases. Indeed, because the stub loads dynamically depending on the particulars of the remote-object implementation, stubs for objects that have the same apparent class might be very different, because the full implementation class of those objects might be very different.

This points out what is perhaps the most fundamental difference between most existing RPC systems and Java RMI. In most existing systems, the result of writing an IDL interface is a static wire protocol, which defines the way the stub of one member of the distributed computation will interact with the skeleton that belongs to another part of the distributed computation. In the RMI system, the interaction point has moved into the address space of the client of a remote object and is defined in terms of a Java interface. That interface's implementation comes from the remote object itself, is dynamically loaded when needed, and can vary in remote objects that appear, from the client's point of view, to be of the same type (because the client only knows that remote objects are of at least some type).

References

1. A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Vol. 2, No. 1, Jan. 1984, pp. 39–59.
2. T.H. Dineen et al., "The Network Computing Architecture and System: An Environment for Developing Distributed Applications," *Proc. Summer Usenix Conf.*, Usenix Assoc., Berkeley, Calif., 1987, pp. 385–398.
3. *Common Object Request Broker: Architecture and Specification, Revision 2.1*, Object Management Group, Framingham, Mass., 1997.
4. *The Component Object Model Specification*, Microsoft, Redmond, Wash.; http://www.microsoft.com/oledev/olecom/title.htm.

Jim Waldo is a senior staff engineer with Sun Microsystems, where he is the lead architect for Jini, a distributed programming infrastructure for Java. He is also an adjunct faculty member of Harvard University's Department of Computer Science, where he teaches distributed computing. He currently works in the area of Java-centric distributed computing. He received his PhD in philosophy from the University of Massachusetts at Amherst and also holds MA degrees in linguistics and philosophy. He edited *The Evolution of C++: Language Design in the Marketplace of Ideas* and writes the "Java Advisor" column for *Unix Review*. He is a member of the IEEE and the ACM. Contact him at Sun Microsystems, 2 Elizabeth Dr., Clemsford, MA 01824-4195; jim.waldo@sun.com.