8 RMI Architecture

The RMI system is designed to provide a direct, simple foundation for distributed object oriented computing. The architecture is designed to allow for future expansion of server and reference types so that RMI can add features in a coherent way.

When a server is exported, its reference type is defined. In the examples above we exported the servers as UnicastRemoteObject servers, which are point-to-point-unreplicated servers. The references for these objects are appropriate for this type of server. Different server types would have different reference semantics. For example, a MulticastRemoteObject would have reference semantics that allowed for a replicated service.

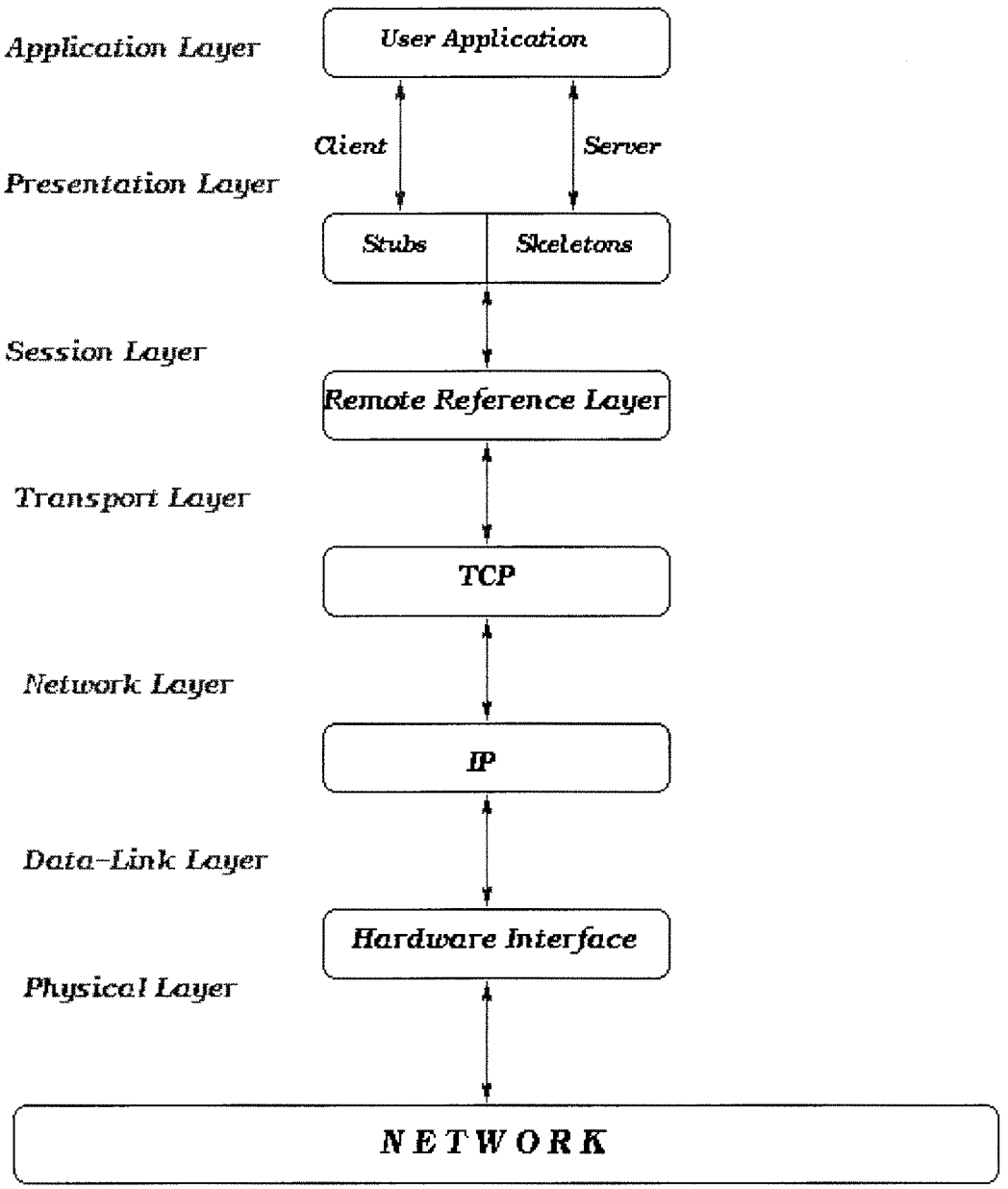
When a client receives a reference to a server, RMI downloads a stub that translates calls on that reference into remote calls to the server. As shown in Figure 3, the stub marshals the arguments to the method using object serialization, and sends the marshalled invocation across the wire to the server. On the server side the call is received by the RMI system and connected to a skeleton, which is responsible for unmarshalling the arguments and invoking the server's implementation of the method. When the server's implementation completes, either by returning a value or by throwing an exception, the skeleton marshals the result and sends a reply to the client's stub. The stub unmarshals the reply and either returns the value or throws the exception as appropriate. Stubs and skeletons are generated from the server implementation, usually using the program rmic. Stubs use references to talk to the skeleton. This architecture allows the reference to define the behavior of communication. The references used for UnicastRemoteObject servers communicate with a single server object running on a particular host and port. With the stub/reference separation RMI will be able to add new reference types. A reference that dealt with replicated servers would multicast server requests to an appropriate set of replicants, gather in the responses, and return an appropriate result based on those multiple responses. Another reference type could activate the server if it was not already running in a virtual machine. The client would work transparently with any of these reference types.

8.1 RMI and the OSI Reference Model

Instead of working directly with Sockets, Client/Server applications can be developed using Java's Remote Method Invocation. Java RMI is a package that can be used to build distributed systems. It allows you to invoke methods on other Java Virtual Machines (possibly on different hosts). The RMI system is very similar to (but more general and easier to use) than the Remote Procedure Call (RPC) mechanisms found on other systems, in that the programmer has the illusion of calling a local method from a local class, where in fact all the arguments are shipped to the remote target, interpreted, and results are sent back to the callers.

One distinguishing aspect of RMI is its simplicity. The set of features supported by RMI are those that are most valuable for building distributed applications, namely: transparent invocations, distributed garbage collection, convenient access to streams. Remote invocations are transparent since they are identical to local ones, so their method signature is identical.

The **OSI Reference Model** defines a framework that consists of seven layers of network communication. The figure below shows how this model can describe RMI.



The User's application is at the top layer, it uses a data representation scheme to transparently communicate with remote objects, possibly on other Java Virtual Machine hosts.

Developing Client/Server applications using sockets involves the design of a protocol that consists of a language agreed upon by the client and server. The design of protocols is hard and error-prone. Using Java Remote Method Invocation or JavaRMI, developing

client-server applications is straight forward because remote invocations in RMI are identical to local ones, so their method signature is identical.

8.2 Safety and Security

There are clear safety and security implications when you are executing RMI requests. RMI provides for secure channels between client and server and the isolation of downloaded implementations inside a security "sandbox" to protect your system from possible attacks by untrusted clients.

First it is important to define your security needs. If you are executing something like the ComputeServer inside a secure corporate network, you may simply need to be able to know who is using the compute cycles so you can track down anyone abusing the system. If you wanted to provide a commercial compute server you would need to protect against more malicious acts. These will affect the exact design of the interface-internally you may just require that each Task object come accompanied by a person's name and department number for tracking purposes. In the commercial case you would want tighter security, including a digitally signed identity and some contractual language that would let you kill off a rogue task that was consuming more than its allotted time.

You may need a secure channel between client and server. RMI lets you provide a socket factory that can create sockets of any type you need, including encrypted sockets. Starting with JDK 1.2, you will be able to specify requirements on the services provided for a server's sockets by giving a description of those requirements. This new technique will work in applets, where most browsers refuse permission to set the socket factory. The socket requirements can include encryption as well as other requirements.

Downloaded classes present security issues as well. Java handles security via a SecurityManager object, which passes judgement on all security-sensitive actions, such as opening files and network connections. RMI uses this standard Java mechanism by requiring that you install a security manager before exporting any server object or invoking any method on a server. RMI provides an RMISecurityManager type that is as restrictive as those used for applets (no file access, only connections to the originating host, and so forth). This will prevent downloaded implementations from reading or writing data from the computer, or connecting to other systems behind your firewall. You

can also write and install your own security manager object to enforce different security constraints.

8.3 Firewalls

RMI provides a means for clients behind firewalls to communicate with remote servers. This allows you to use RMI to deploy clients on the Internet, such as in applets available on the World Wide Web. Traversing the client's firewall can slow down communication, so RMI uses the fastest successful technique to connect between client and server. The technique is discovered by the reference for UnicastRemoteObject on the first attempt the client makes to communicate with the server by trying each of three possibilities in turn:

- Communicate directly to the server's port using sockets.
- If this fails, build a URL to the server's host and port and use an HTTP POST request on that URL, sending the information to the skeleton as the body of the POST. If successful, the results of the post are the skeleton's response to the stub.
- If this also fails, build a URL to the server's host using port 80, the standard HTTP port, using a CGI script that will forward the posted RMI request to the server.

Whichever of these three techniques succeeds first is used for all future communication with the server. If none of these techniques succeeds, the remote method invocation fails. This three-stage back off allows clients to communicate as efficiently as possible, in most cases using direct socket connections. On systems with no firewall, or with communication inside an enterprise behind a firewall, the client will directly connect to the server using sockets. The secondary communication techniques are significantly slower than direct communication, but their use enables you to write clients that can be used broadly across the Internet and Web.

8.4 RMI in an Evolving Enterprise

You can use RMI today to connect between new Java applications (or applets) and existing servers. When you do this, you allow your organization to benefit incrementally from expanded Java use over time. When parts of your systems are rewritten in Java,

RMI allows the benefits of Java to flow from the existing Java components into the new Java code. Consider the path of a single request in a two-tier system from the client to the server and back again

Using RMI means that you can get Java benefits throughout your system by using RMI as the transport between client and server, even if the server remains in non-Java code for some time. If you choose to rewrite some or all of your servers in Java, you will get leverage from your existing Java components. Some of the most important Java advantages you maintain are:

- Object oriented code reuse. The ability to pass objects from client to server and server to client means that you can use design patterns and other object oriented programming techniques to enhance code reuse in your organization.
- Passing behavior. The objects passed between client and server can be of types not previously seen by the other side. Implementations will be downloaded to execute the new behavior.
- Type safety. Java objects are always type safe, preventing bugs that would occur if a programmer makes a mistake about the type of an object.
- Security. Java Classes can be run in a secure fashion, allowing you to interact with clients that may be running in an untrusted environment. Here is that diagram showing a client written in Java using RMI to talk to the server. The "request" arrow has been shaded to show where you get Java's safety, object oriented, and other advantages:

A small amount of Java code connects to a "legacy wrapper" that uses the existing server's API. The legacy wrapper is the bridge between Java and the existing API, as shown in the implementations of getUnpaid and shutDown above. In this diagram we show it written using JNI, but as shown above the legacy wrapper could use JDBC or, when it is available, TwinPeaks.

Contrast the above diagram with one in which a language neutral system using an interface definition language (commonly called an IDL) introduces a least-common-denominator connection between the client and server

A legacy wrapper must still be written to connect the IDL-defined calls to the existing server API. But with an IDL-based approach the benefits of Java have been isolated on the client side-because the client's Java is reduced to the least common denominator before crossing to the server. Suppose you decide that rewriting some of the server in Java would be useful to you. This might be for any reason, such as wanting the improved reliability of a safe Java system, or because you want to reduce porting costs. Or possibly the vendor from whom you bought some of your server Implementation has provided an upgrade that takes advantage of Java. Here is how the RMI based picture now looks:

More of the request now benefits from having Java. The objects that you pass across the wire between client and server can now have more benefits to the overall system. You could, for example, start passing behavior across the same remote interfaces you have already defined to enhance the value of client and server. Compare this to the IDL-based approach:

You would have achieved benefits of Java that are local to the server, but you cannot leverage expanded value of Java for the client-server connection. The benefits of Java are cut off at the IDL boundary because IDL cannot assume that Java is on the other side of the connection. You cannot get the full benefits of Java in your system without throwing out the IDL work and rewriting it using RMI.
This lost opportunity becomes greater as you use Java in more of your enterprise. Using RMI you get benefits of Java all the way through the system:

Using an IDL-based approach, rewriting the server in Java still only gives you localized benefit isolated to the server alone:

You can use RMI today to connect to your servers without rewriting it in Java. RMI is simple to use, so it is easy to create the server-side RMI class. The legacy wrapper is of similar complexity in either case. But if you use an IDL-based distributed system you will create isolated pockets of Java that share no benefits with each other. RMI lets you

connect easily now, and as you decide to expand your use of Java, you will get expanded benefits from the synergy of having Java on both sides of the wire.

8.5 Conclusion on RMI

RMI provides a solid platform for truly object oriented distributed computing. You can use RMI to connect to Java components or to existing components written in other languages. As Java proves itself in your environment, you can expand your Java use and get all the benefits-no porting, low maintenance costs, and a safe, secure environment. RMI gives you a platform to expand Java into any part your system in an incremental fashion, adding new Java servers and clients when it makes sense. As you add Java, its full benefits flow through all the Java in your system. RMI makes this easy, secure, and powerful.