- Java
- Technical Details

Java Remote Method Invocation Distributed Computing for Java

Java Remote Method Invocation Distributed Computing for Java

Overview

Java Remote Method Invocation (RMI) allows you to write distributed objects using Java. This paper describes the benefits of RMI, and how you can connect it to existing and legacy systems as well as to components written in Java.

RMI provides a simple and direct model for distributed computation with Java objects. These objects can be new Java objects, or can be simple Java wrappers around an existing API. Java embraces the "Write Once, Run Anywhere model. RMI extends the Java model to be run everywhere."

Because RMI is centered around Java, it brings the power of Java safety and portability to distributed computing. You can move behavior, such as agents and business logic, to the part of your network where it makes the most sense. When you expand your use of Java in your systems, RMI allows you to take all the advantages with you.

RMI connects to existing and legacy systems using the standard Java native method interface JNI. RMI can also connect to existing relational database using the standard JDBC?? package. The RMI/JNI and RMI/JDBC combinations let you use RMI to communicate today with existing servers in non-Java languages, and to expand your use of Java to those servers when it makes sense for you to do so. RMI lets you take full advantage of Java when you do expand your use.

Advantages

At the most basic level, RMI is Java's remote procedure call (RPC) mechanism. RMI has several advantages over traditional RPC systems because it is part of Java's object oriented approach. Traditional RPC systems are language-neutral, and therefore are essentially least-common-denominator systems-they cannot provide functionality that is not available on all possible target platforms.

RMI is focused on Java, with connectivity to existing systems using native methods. This means RMI can take a natural, direct, and fully-powered approach to provide you with a distributed computing technology that lets you add Java functionality throughout your system in an incremental, yet seamless way.

The primary advantages of RMI are:

- Object Oriented: RMI can pass full objects as arguments and return values, not just predefined data types. This means that you can pass complex types, such as a standard Java hashtable object, as a single

argument. In existing RPC systems you would have to have the client decompose such an object into primitive data types, ship those data types, and the recreate a hashtable on the server. RMI lets you ship objects directly across the wire with no extra client code.

- Mobile Behavior: RMI can move behavior (class implementations) from client to server and server to client. For example, you can define an interface for examining employee expense reports to see whether they conform to current company policy. When an expense report is created, an object that implements that interface can be fetched by the client from the server. When the policies change, the server will start returning a different implementation of that interface that uses the new policies. The constraints will therefore be checked on the client side-providing faster feedback to the user and less load on the server-without installing any new software on user's system. This gives you maximal flexibility, since changing policies requires you to write only one new Java class and install it once on the server host.
- Design Patterns: Passing objects lets you use the full power of object oriented technology in distributed computing, such as two- and three-tier systems. When you can pass behavior, you can use object oriented design patterns in your solutions. All object oriented design patterns rely upon different behaviors for their power; without passing complete objects-both implementations and type-the benefits provided by the design patterns movement are lost.
- Safe and Secure: RMI uses built-in Java security mechanisms that allow your system to be safe when users downloading implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.
- Easy to Write/Easy to Use: RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed

object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation. And because RMI programs are easy to write they are also easy to maintain.

- Connects to Existing/Legacy Systems: RMI interacts with existing systems through Java's native method interface JNI. Using RMI and JNI you can write your client in Java and use your existing server implementation. When you use RMI/JNI to connect to existing servers you can rewrite any parts of you server in Java when you choose to, and get the full benefits of Java in the new code. Similarly, RMI interacts with existing relational databases using JDBC without modifying existing non-Java source that uses the databases.
- Write Once, Run Anywhere: RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine *, as is an RMI/JDBC system. If you use RMI/JNI to interact with an existing system, the code written using JNI will compile and run with any Java virtual machine.
- Distributed Garbage Collection: RMI uses its distributed garbage collection feature to collect remote server objects that are no longer referenced by any clients in the network. Analogous to garbage collection inside a Java Virtual Machine, distributed garbage collection lets you define server objects as needed, knowing that they will be removed when they no longer need to be accessible by clients.
- Parallel Computing: RMI is multi-threaded, allowing your servers to exploit Java threads for better concurrent processing of client requests.
- The Java Distributed Computing Solution: RMI is part of the core Java platform starting with JDK?? 1.1, so it exists on every 1.1 Java Virtual Machine. All RMI systems talk the same public protocol, so all Java systems can talk to each other directly, without any protocol translation overhead.

Passing Behavior

When we described how RMI can move behavior above, we briefly outlined an expense report program. Here is a deeper description of how you could design such a system. We present this to show how you can use RMI's ability to move behavior from one system to another to move computing to where you want it today, and change it easily tomorrow. The examples below do not handle all cases that would arise in the real world, but instead give a flavor for how the problem can be approached.

Server-Defined Policy

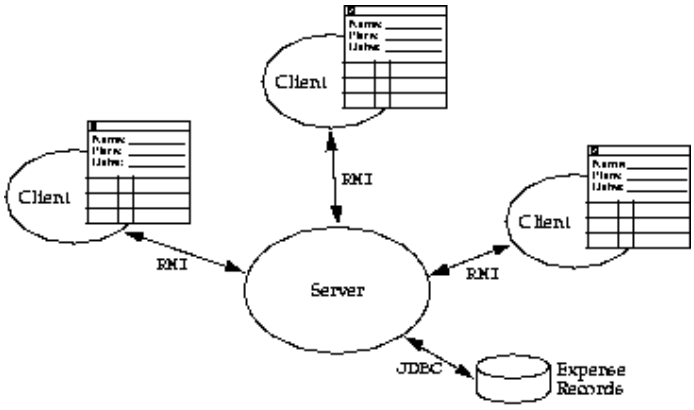


FIGURE 1 An Expense Reporting Architecture

Figure 1 shows the general picture of such a dynamically configurable expense reporting system. A client displays a GUI (graphical user interface) to a user, who fills in the fields of the expense report. Clients communicate with the server using RMI. The server stores the expense reports in a database using JDBC, the Java relational database package. So far this may look like any multi-tier system, but there is an important difference-RMI can download behavior.

Suppose that the company's policies about expense reports change. For example, today the company requires receipts only for expenses over $20. Tomorrow the company decides this is too lenient-it wants receipts for everything, except for meals that cost less than $20. Without the ability to download behavior, you have the following alternatives when designing your system for change:

- Install the policy with the client. When the policy changes, this requires updating all clients that contain the policy. You could reduce the problem by installing the client on a handful of server machines and requiring all users to run the client from one of those servers. This still would not completely solve the problem-anyone who leaves the program up and runn ing for days would not be updated, and there are always some people who copy the software to a local disk for efficiency.
- You could have the policy checked by the server when each entry is added to the expense report. This would result in a lot of traffic between client and server, clogging the network and burdening the server. It would also make the system more fragile-a network failure would halt people in their tracks instead of only affecting them when they actually submit an expense report or start a new one. It would also mean that adding an entry would be slow, since it would require a round trip across the network to the (burdened) server.
- You could have the policy checked by the server when the report is submitted. This lets the user create a lot of bad entries which must then

be reported in a batch instead of catching the first error immediately, giving the user a chance to stop making the error. Users need immediate feedback on errors to avoid wasted time.

With RMI you can have the client upload behavior from the server with a simple method invocation, providing a flexible way to offload computation from the server to the clients while providing users with faster feedback. When a user is ready to write up a new expense report, the client asks the server for an object that embodies the current policies for expense reports as expressed via a Policy interface written in Java. The object can implement the policy in any way. If this is the first time that the client's RMI runtime has seen this particular implementation of the policy, RMI will ask the server for a copy of the implementation. Should the implementation change tomorrow, a new kind of policy object will be returned to the client, and the RMI runtime will then ask for that new implementation.

This means that policy is always dynamic. You can change the policy by simply writing a new implementation of the general Policy interface, installing it on the server, and configuring the server to return objects of this new type. From that point on, any new expense reports will be checked against the new policy by every client.

This is a better approach than any static approach because:

- All clients don't need to be halted and updated with new software-software is updated on the fly as needed.
- The server is not burdened with entry checking that can be done locally.
- Allows dynamic constraints because object implementations, not just data, are passed between client and server.
- Lets users know immediately about errors.

Here is the remote interface that defines the methods the client can invoke on the server:

```
import java.rmi.*;
public interface ExpenseServer extends Remote {
    Policy getPolicy() throws RemoteException;
    void submitReport(ExpenseReport report)
        throws RemoteException, InvalidReportException;
}
```

The import statement imports the Java RMI package. All the RMI types are defined in the package java.rmi or one of its subpackages. The interface ExpenseServer is a normal Java interface with two interesting characteristics

- It extends the RMI interface named Remote, which marks the interface as one available for remote invocation.
- All its methods throw RemoteException, which is used to signal network and messaging failures. Remote methods can throw any other exception you like, but they must throw at least RemoteException so that you can handle error conditions that only arise in distributed systems. The interface itself supports two methods: getPolicy which returns an object that implements the Policy interface, and

submitReport which submits a completed expense request, throwing an exception if the report is malformed for any reason.

The Policy interface itself declares a method that lets the client know if it is acceptable to add an entry to the expense report:

```
public interface Policy {
    void checkValid(ExpenseEntry entry)
        throws PolicyViolationException;
}
```

If the entry is a valid one-one that matches current policy-the method returns normally. Otherwise it throws an exception that describes the error. The Policy interface is local (not remote), and so will be implemented by an object local to the client-one that runs in the client's virtual machine, not across the network. A client would operate something like this:

```
Policy curPolicy = server.getPolicy();
start a new expense report
show the GUI to the user
while (user keeps adding entries) {
    try {
        curPolicy.checkValid(entry); // throws exception if not OK
        add the entry to the expense report
    } catch (PolicyViolationException e) {
        show the error to the user
    }
}
server.submitReport(report);
```

When the user asks the client software to start up a new expense report, the client invokes server.getPolicy to ask the server to return an object that embodies the current expense policy. Each entry that is added is first submitted to that policy object for approval. If the policy object reports no error, the entry is added to the report; otherwise the error will be displayed to the user who can take corrective action. When the user is finished adding entries to the report, the entire report is submitted. The server looks like this:

```
import java.rmi.*;
import java.rmi.server.*;
class ExpenseServerImpl
    extends UnicastRemoteObject
    implements ExpenseServer
{
    ExpenseServerImpl() throws RemoteException {
        // ...set up server state...
    }
    public Policy getPolicy() {
        return new TodaysPolicy();
    }
    public void submitReport(ExpenseReport report) {
        // ...write the report into the db...
    }
}
```

We import RMI's server package in addition to the basic package. The type UnicastRemoteObject defines the kind of remote object this server will be, in this case a single server as opposed to a replicated service (more on this later). The Java class ExpenseServerImpl implements the methods of the remote interface ExpenseServer. Clients on remote hosts can use RMI to send messages to ExpenseServerImpl objects.

The important method for this discussion is getPolicy, which simply returns an object that defines the current policy. Let's take a look at an example implementation of a policy:

```
public class TodaysPolicy implements Policy {
    public void checkValid(ExpenseEntry entry)
        throws PolicyViolationException
```

```
    {
        if (entry.dollars() < 20) {
            return; // no receipt required
        } else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException;
        }
    }
}
```

TodaysPolicy checks to ensure that any entry without a receipt is less than $20. If the policy changes tomorrow so that only meals under $20 are exempt from the "receipts required" policy, you could provide a new implementation of policy:

```
public class TomorrowsPolicy implements Policy {
    public void checkValid(ExpenseEntry entry)
        throws PolicyViolationException
    {
        if (entry.isMeal() && entry.dollars() < 20) {
            return; // no receipt required
        } else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException;
        }
    }
}
```

Write this class, install it on the server, and tell the server to start handing out TomorrowsPolicy objects instead of TodaysPolicy objects, and your entire system will start using the new policy. When the client invokes the server's getPolicy method, RMI on the client checks to see if the returned object is of a known type. The first time each client encounters a TomorrowsPolicy object, RMI will download the implementation for the policy before getPolicy returns. The client will, without effort, start enforcing the new policy.

RMI uses the standard Java object serialization mechanism to pass objects. Arguments that are references to remote objects are passed as remote references. If an argument to a method is a primitive type or a local (non-remote) object, a deep copy is passed to the server. Return values are handled in the same way, but in the other direction. RMI lets you pass and return full object graphs for local objects and references to remote objects.

In a real system the getPolicy method might have a parameter that identified the user and the kind of expense report (travel, customer relations, etc.) so that the policy can differ. Or instead of requiring separate policy and expense report object, you might have a newExpenseReport method that returned an ExpenseReport object that directly checked the policy. This last strategy would allow you to change the contents of an expense report as easily as the policy-when the company decides that it needs to split out meals into separate breakfast, lunch, and dinner entries that change would be implemented as easily as the new policy shown above-write a new class implementing the report and the client will use it automatically.

Compute Server

The expense report example shows how the client can get behavior from the server. Behavior can flow in both directions-the client can equally pass new types to the user. The simplest example of this would be a compute server such as that shown in Figure 2, where the server is available to execute arbitrary tasks so that clients all across the enterprise can take advantage of high-end or specialized computers.

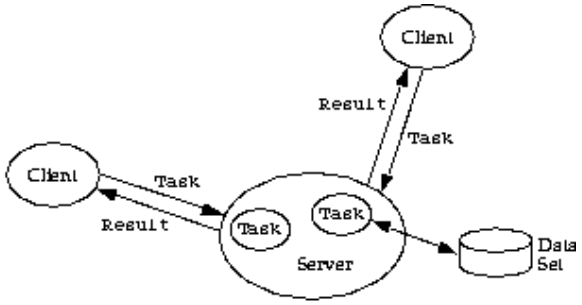


FIGURE 2 An Compute Server Architecture

Tasks are defined by a simple local (non-remote) interface:

```
public interface Task {
    Object run();
}
```

When run is invoked, it does some computation and returns an object that contains the results. This is purposefully completely generic-almost any computational task can be implemented under this interface. The remote interface ComputeServer is equally simple:

```
import java.rmi.*;
public interface ComputeServer extends Remote {
    Object compute(Task task) throws RemoteException;
}
```

The only purpose of this remote interface is to allow a client to create a Task object and send it to the server for execution, returning the results. Here is a basic implementation of that server:

```
import java.rmi.*;
import java.rmi.server.*;
public class ComputeServerImpl
    extends UnicastRemoteObject
    implements ComputeServer
{
    public ComputeServerImpl() throws RemoteException { }
    public Object compute(Task task) {
        return task.run();
    }
    public static void main(String[] args) throws Exception {
        // use the default, restrictive security manager
        System.setSecurityManager(new RMISecurityManager());
        ComputeServerImpl server = new ComputeServerImpl();
        Naming.rebind("ComputeServer", server);
        System.out.println("Ready to receive tasks");
        return;
    }
}
```

If you look at the compute method you will see that it, too, is very simple. It just takes the object that is passed to it and returns the results of whatever computation run performs. The main method contains the startup code for the server-it installs RMI's default security manager to prevent any access to the local system, creates a ComputeServerImpl object for handling incoming requests, and then binds it to the name "ComputeServer". At this point the server is ready to receive tasks to execute, and main is finished with its setup.

As described this is actually a complete and useful service. The system could be enhanced, such as by adding parameters to compute so that departments can be billed back for their use of the server. But in many situations the interface and implementation described allows a high-end computer to be used for remote computation. This is a powerful demonstration of the simplicity of RMI-if you type in the above classes, compile them, and start up a server, you will have a running compute server capable of executing arbitrary tasks.

Let's walk through an example of using this compute service. Assume that a very high-end system is purchased so that compute-intensive applications can be run. An administrator would start up a Java virtual machine on that system, running a ComputeServerImpl object. That object would now be ready to accept tasks to run.

Now suppose that a group wants to train a neural network against a set of data in order to help plan purchasing strategy. Here are the steps they would go through:

- Define a class-call it PurchaseNet-that takes a set of data and runs training data through it, returning a trained neural network. PurchaseNet would implement the Task interface, executing its work in its run method. They would need a Neuron class to describe the nodes in the network that is returned, and probably other classes that would describe the processing. The run method would return a NeuralNet object that was a collection of trained Neuron objects.
- When these class are written and work on a small test case, invoke the compute method of the ComputeServer with a PurchaseNet object.
- When the RMI system in ComputeServerImpl process receives the PurchaseNet object as an incoming parameter, it downloads the implementation for PurchaseNet and then invokes the server's compute method with that object as the Task argument.
- The Task, being a PurchaseNet object, will start executing its implementation. When the implementation requires new classes, such as Neuron and NeuralNet they will be downloaded as needed.
- All the computation will be executed on the compute server while the client thread awaits the results. (Another thread on the client system could be displaying a "waiting" cursor or performing another task using Java's built-in concurrency mechanisms.) When run returns its NeuralNet object, that will be passed back to the client as the result of the compute method.

This requires no installation of additional software on the server-everything needed is done by the department on its own machines, and then made available to the compute server host when needed.

The simple compute server infrastructure provides a powerful shift in your systems distributed power. The computation of a task can be moved to a system that can best support it. Equivalent systems could be used to:

- Support a data mining application where the ComputeServerImpl object is run on the host with the data that needs to be mined. This lets you easily move any computation to where the data is.

- Run computation on a server with direct data feed from an outside source, such as current stock prices, shipping information, or other realtime information.
- Distribute tasks across multiple servers by having a different implementation of ComputeServer that took incoming requests and forwarded them to the least-burdened server running a ComputeServerImpl.

Agents

Because RMI lets you download behavior using Java implementations, you can write an agent system using RMI. In its simplest form it would look like this:

```
import java.rmi.*;
public interface AgentServer extends Remote {
    void accept(Agent agent)
        throws RemoteException, InvalidAgentException;
}
public interface Agent extends java.io.Serializable {
    void run();
}
```

Starting an agent would be a matter of creating a class that implemented the Agent interface, finding a server, and invoking accept with that agent object. The implementation for the agent would be downloaded to the server and run there. The accept method would start up a new thread for the agent, invoke its run method, and then return, letting the agent continue execution after the method returned. The agent could migrate to another host by invoking accept on a server running on that host, passing itself as the agent to be accepted, and then killing its thread on the original host.

Object Oriented Code Reuse and Design Patterns

Object oriented programming is a powerful technique for allowing code reuse. Many organizations are using object oriented programming to reduce the burden of creating programs and to increase the flexibility of their systems. RMI is object oriented at all levels-messages are sent to remote objects, and objects can be passed and returned.

The Design Patterns movement has been very successful in describing good practices in object oriented design. First made popular by the seminal work Design Patterns, these patterns of programming are a way to formally describe an overall approach to a particular kind of problem. All of these design patterns rely upon creating one or more abstractions that allow varying implementations, thereby enabling and enhancing software reuse. Software reuse is one of the core promises of object oriented technology, and design patterns are one of the most popular techniques for promoting reuse.

All design patterns rely upon object oriented polymorphism-the ability of an object (such as Task) to have multiple implementations. The general part of the algorithm (such as the compute method) does not need to know which particular implementation is present, it need only know what to do with such an object when it gets one. In particular, the compute server is an example of the Command pattern, which lets you represent a request (task) as an object, letting it be dispatched.

This polymorphism is only available if the full objects, including implementations, can be passed between client and server. Traditional RPC systems, such as DCE and DCOM, and object based RPC systems, such as CORBA, cannot download and execute implementations because they cannot pass real objects as arguments, only data.

RMI passes full types, including implementations, so you can use object oriented programming-including design patterns-every where in your distributed computing solutions, not just in local computation. Without RMI's fully object oriented system, you need to abandon design patterns-along with other forms of object oriented software reuse-in much of your distributed computing systems.

Connecting To An Existing Server

It is commonly said that RMI works for "Java to Java," but this hides an important fact: Java connects quite well to existing and legacy systems using the native method interface called JNI. You can use JNI with RMI as easily as with any other Java code. You can use JDBC with RMI to connect to existing relational databases. This means you can use RMI to connect two- and three-tier systems even if both sides are not written in Java. There are significant advantages to doing so, as we will discuss below. But first let's see how it can be done.

Let us suppose that you had an existing server that stored information on customer orders in a relational database. In any multi-tier system you would have to design a remote interface to let clients access the server-with RMI that would be a Remote interface:

```
import java.rmi.*;
import java.sql.SQLException;
import java.util.Vector;
public interface OrderServer extends Remote {
    Vector getUnpaid() throws RemoteException, SQLException;
    void shutDown() throws RemoteException;
    // ... other methods (getOrderNumber, getShipped, ...)
}
```

The package java.sql contains the JDBC package. Each remote method can be implemented by the server using JDBC calls onto the actual database or by native methods that use other database access mechanisms. The methods shown return a vector (list) of Order objects. Order is a class defined in your system to hold a customer's order.

In this section we will show how you can implement getUnpaid using JDBC and shutDown using JNI.

JDBC-Direct to the Database

Here is OrderServerImpl that uses JDBC to implement getUnpaid:

```
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;
import java.util.Vector;
public class OrderServerImpl
    extends UnicastRemoteObject
    implements OrderServer
{
    Connection db;                  // connection to the db
    PreparedStatement unpaidQuery;  // unpaid order query
    OrderServerImpl() throws RemoteException, SQLException {
        db = DriverManager.getConnection("jdbc:odbc:orders");
        unpaidQuery = db.prepareStatement("...");
    }
    public Vector getUnpaid() throws SQLException {
        ResultSet results = unpaidQuery.executeQuery();
        Vector list = new Vector();
        while (results.next())
            list.addElement(new Order(results));
        return list;
    }
```

```
    public native void shutDown();
}
```

Most of this is JDBC work. All types you see are part of JDBC or RMI except those that start with Order, which are part of your system. The constructor initializes the OrderServerImpl object, creating a Connection to a database that is specified in a jdbc URL. Given that connection, we use prepareStatement to define a query that will find all unpaid orders. Other queries could be defined here for other methods. The OrderServerImpl server runs on the same system as the database, possibly in the same process. (We will describe shutDown below).

When a getUnpaid method is invoked on the RMI server object OrderServerImpl, the precompiled query is executed, returning a JDBC ResultSet object that contains all the matching elements. We then create new Order objects for each item in the result set, putting each into a Vector object (Java's dynamically-sized array). When we are finished reading the results, we return the Vector to the client, who can then display the results to the user or do whatever else is appropriate.

JNI-Native Methods

RMI servers and clients can use native methods as a bridge to existing and legacy systems. You can use native methods to implement remote methods that are not direct database access, or which can be implemented more easily using existing code. The native interface JNI lets you write C and C++ code that implements Java methods and invokes methods on Java objects. Here is how shutDown could be implemented by a native method:

```
JNIEXPORT void JNICALL
Java_OrderServerImpl_shutDown(JNIEnv *env, jobject this)
{
    jclass cls;
    jfieldID fid;
    DataSet *ds;
    cls = (*env)->GetObjectClass(env, this);
    fid = (*env)->GetFieldID(env, cls, "dataSet", "J");
    ds = (DataSet *) (*env)->GetObjectField(env, this, fid);
    /* With a DataSet pointer we can use the original API */
    DSshutDown(ds);
}
```

This presumes that the existing server is referenced via a DataSet type defined by the existing server's API. A pointer to server's DataSet would be stored in a dataSet field. When the client invokes shutDown, this causes the shutDown method of the server to be invoked. Because the server implementation declares shutDown to be implemented with a native method, RMI will directly invoke this native method. The native method finds the dataSet field of the object, gets its value, and uses it to invoke the existing API function DSshutDown.

Sun is currently working in partnership with ILOG on a product called TwinPeaks. TwinPeaks will take existing C and C++ APIs and generate Java classes that wrap calls to that API in Java classes. This would allow you to invoke methods on arbitrary existing APIs from Java. When TwinPeaks is available it will make it possible to write methods like shutDown completely in Java instead of using JNI calls.

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.

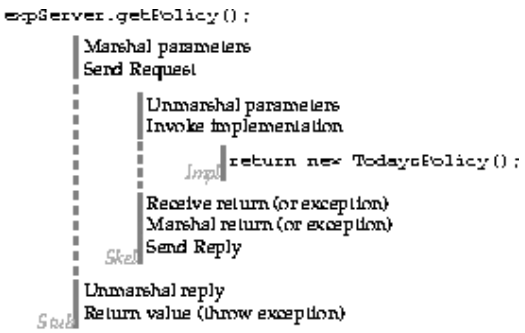


FIGURE 3 Stubs and Skeletons

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.

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.

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:

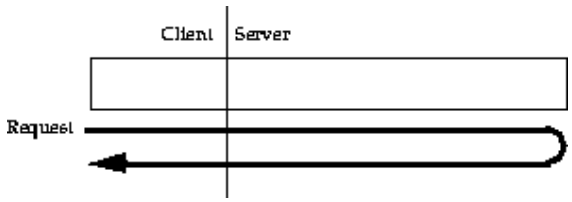
- Communicate directly to the server's port using sockets.
- If this fails, build a URL to the server's host and port and use an HTTP POST request on that URL, sending the information to the skeleton as the body of the POST. If successful, the results of the post are the skeleton's response to the stub.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

RMI in an Evolving Enterprise

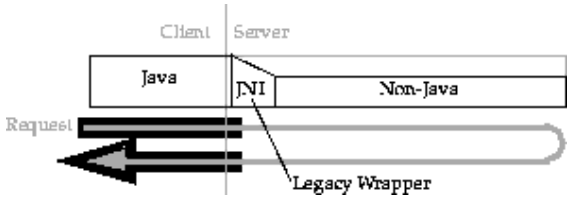
You can use RMI today to connect between new Java applications (or applets) and existing servers. When you do this, you allow your organization to benefit incrementally from expanded Java use over time. When parts of your systems are rewritten in Java, RMI allows the benefits of Java to flow from the 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: [1]



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:

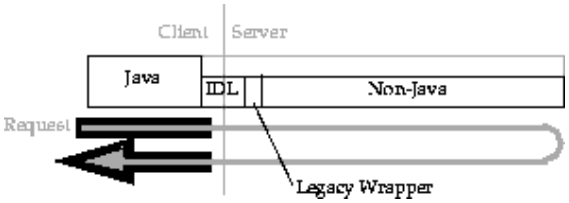
- Object oriented code reuse. The ability to pass objects from client to server and server to client means that you can use design patterns and other object oriented programming techniques to enhance code reuse in your organization.

- 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:

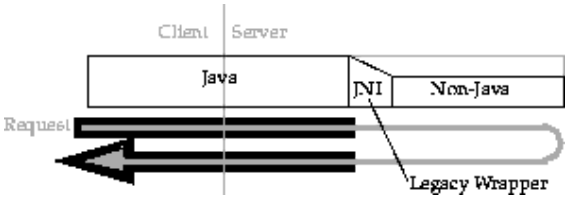


A small amount of Java code connects to a "legacy wrapper" that uses the existing server's API. The legacy wrapper is the bridge between Java and the existing API, as shown in the implementations of getUnpaid and shutDown above. In this diagram we show it written using JNI, but as shown above the legacy 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: [2]

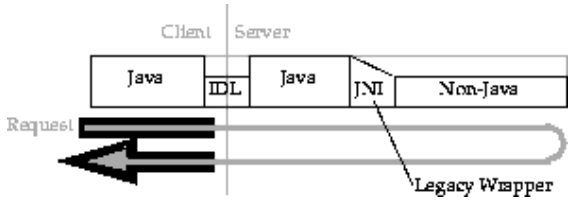


A legacy wrapper must still be written to connect the IDL-defined calls to the existing server API. But with an IDL-based approach the benefits of Java have been isolated on the client side-because the client's Java is reduced to the least common denominator before crossing to the server. Suppose you 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

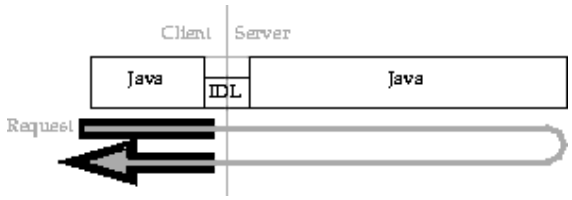
remote interfaces you have already defined to enhance the value of client and server. Compare this to the IDL-based approach:



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.

Conclusion

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.

***As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.**

Resources for Why Oracle Learn What's New Contact Us

Careers Analyst Reports

Developers
Investors
Partners
Startups
Students and Educators

Gartner MQ for Cloud ERP
Cloud Economics
Corporate Responsibility
Diversity and Inclusion
Security Practices

What is cloud computing?
What is CRM?
What is Docker?
What is Kubernetes?
What is Python?
What is SaaS?

Oracle Supports Ukraine
Oracle CloudWorld
Oracle and Premier League
Oracle Red Bull Racing
Employee Experience Platform
Oracle Support Rewards

US Sales: +1.800.633.0738
How can we help?
Subscribe to emails
Events
News
Blogs

© 2022 Oracle
Site Map
Privacy / Do Not Sell My Info
Cookie Preferences
Ad Choices
Careers
Country/Region