**The Java™ Tutorials**

An Overview of RMI Applications

RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

Distributed object applications need to do the following:

- **Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- **Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- **Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.
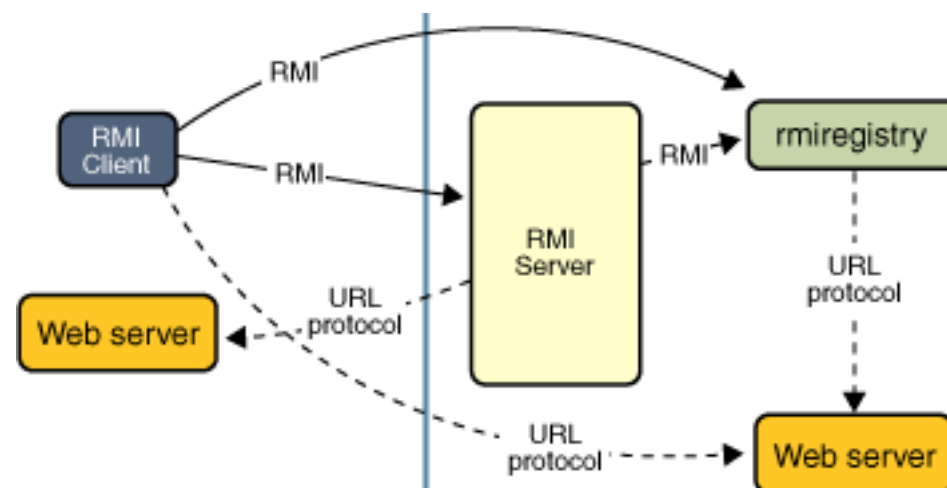
Advantages of Dynamic Code Loading

One of the central and unique features of RMI is its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine. All of the types and behavior of an object, previously available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine. RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine. This capability enables new types and behaviors to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application. The compute engine example in this trail uses this capability to introduce new behavior to a distributed program.

Remote Interfaces, Objects, and Methods

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called *remote objects*.

An object becomes remote by implementing a *remote interface*, which has the following characteristics:

- A remote interface extends the interface `java.rmi.Remote`.
- Each method of the interface declares `java.rmi.RemoteException` in its `throws` clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote *stub* for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, *only* those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Creating Distributed Applications by Using RMI

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

Designing and Implementing the Application Components

First, determine your application architecture, including which components are local objects and which components are remotely accessible. This step includes:

- **Defining the remote interfaces.** A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
- **Implementing the remote objects.** Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.
- **Implementing the clients.** Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Compiling Sources

As with any Java program, you use the `javac` compiler to compile the source files. The source files contain the declarations of the remote interfaces, their implementations, any other server classes, and the client classes.

Note: With versions prior to Java Platform, Standard Edition 5.0, an additional step was required to build stub classes, by using the `rmic` compiler. However, this step is no longer necessary.

Making Classes Network Accessible

In this step, you make certain class definitions network accessible, such as the definitions for the remote interfaces and their associated types, and the definitions for classes that need to be downloaded to the clients or servers. Classes definitions are typically made network accessible through a web server.

Starting the Application

Starting the application includes running the RMI remote object registry, the server, and the client.

The rest of this section walks through the steps used to create a compute engine.

Building a Generic Compute Engine

This trail focuses on a simple, yet powerful, distributed application called a *compute engine*. The compute engine is a remote object on the server that takes tasks from clients, runs the tasks, and returns any results. The tasks are run on the machine where the server is running. This type of distributed application can enable a number of client machines to make use of a particularly powerful machine or a machine that has specialized hardware.

The novel aspect of the compute engine is that the tasks it runs do not need to be defined when the compute engine is written or started. New kinds of tasks can be created at any time and then given to the compute engine to be run. The only requirement of a task is that its class implement a particular interface. The code needed to accomplish the task can be downloaded by the RMI system to the compute engine. Then, the compute engine runs the task, using the resources on the machine on which the compute engine is running.

The ability to perform arbitrary tasks is enabled by the dynamic nature of the Java platform, which is extended to the network by RMI. RMI dynamically loads the task code into the compute engine's Java virtual machine and runs the task without prior knowledge of the class that implements the task. Such an application, which has the ability to download code dynamically, is often called a *behavior-based application*. Such applications usually require full agent-enabled infrastructures. With RMI, such applications are part of the basic mechanisms for distributed computing on the Java platform.

Problems with the examples? Try Compiling and Running the Examples: