How Java RMI Works (in 5 Pages or Less)

By: André Campeau

Java RMI (Remote Method Invocation) can be used to invoke methods of objects in another Java Virtual Machine (JVM). The JVM can be on the local host or another host. RMI allows the client (the thread that actually invokes a method in the remote object) to invoke remote methods in the server (the thread that contains the remote object) as if the remote object were contained on the client host, from the programmer's perspective.

What really happens is that when a client calls a method in a remote object, instead of letting the Java runtime take care of the call as it would for regular Java objects, the RMI runtime system takes over and routes the call over the network to the remote object. In essence, the RMI runtime opens a socket connection on an anonymous port to the remote object running on the server host. The RMI runtime (running as part of the JVM on the server side) listens on this special port for incoming RMI requests and invokes the requested method on the server side when one is received. Information returned by the called method is passed back to the client through the socket, after which the socket is closed. None of the underlying communication is visible to the programmer.

As a programmer, one may wish to access remote objects from one or more servers out on the network. The remote objects could, for example, contain methods which, when invoked, return information about the hardware or software running on the server (sound familiar?). These remote "agent" objects may exist on many servers on the network (here "server" refers to a network element which contains remote objects whose methods can be accessed by interested "clients" using RMI). So how does a client, which may come into existence at any time, find out what remote objects it can access on servers on the network? The answer is: by getting the server object reference from the remote object registry.

The remote object registry runs on the server where the remote object exists. It contains a list of references to all remote objects on that server that are accessible to clients through RMI. A client may query the remote object registry (henceforth known as the "registry") to obtain a reference to a remote object, and set a local object's reference to that of the remote object. For example, if the object `obj` was defined in the Java client program, the

registry could be queried (by specifying the URL of the remote object) to find the reference to the object `remoteObj`, and `obj` could be set to point to `remoteObj`. Calls to methods in `obj` would be intercepted by the RMI runtime and redirected to `remoteObj` on the server using socket communication as explained previously. Note that both `obj` and `remoteObj` must be instances of the same class, or more specifically, `obj` must implement the interface of the class that `remoteObj` is instantiated from (more on this later).

This explains how to access remote methods from a client, but how does one set up methods on a server so that they can be accessed remotely? Each remote object server host computer must be running a registry. The registry may be running in the same JVM as the remote object, or in a separate JVM. The registry is an autonomous Java thread that responds to requests from other Java programs on the network using socket communication. The registry listens on port 1099 by default for incoming requests. This port can be changed, but all clients will have to be updated accordingly. This way, you could have more than one registry running on the same server - a necessity if the registry is built-in to more than one remote object on the server.

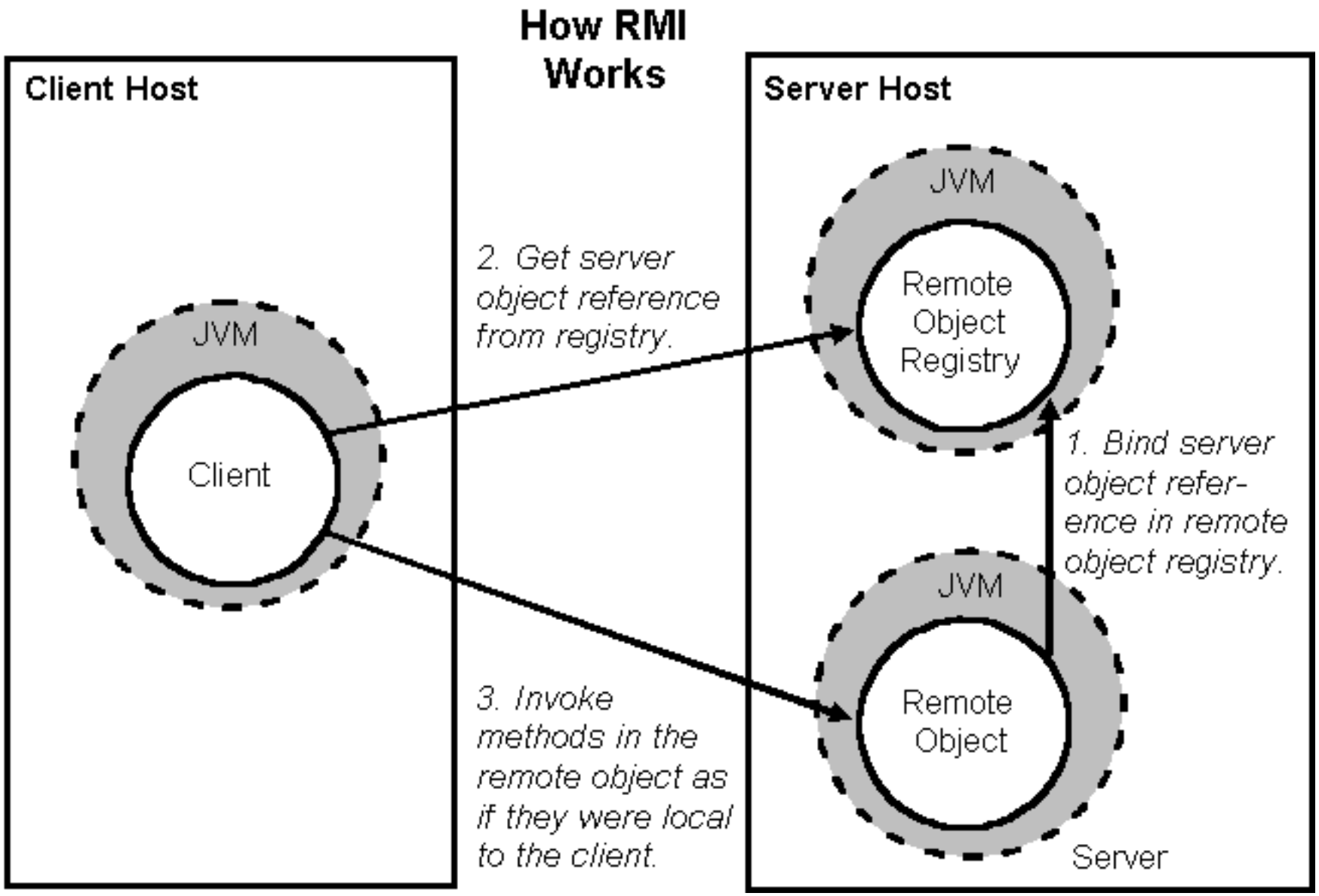
Before any objects become remotely accessible to clients, they must be registered in the registry (a process called "binding"). Objects may only be bound in a registry on the same host. This means that you will never need to have a registry running on a client that is not making any of its objects remotely accessible. If there are many servers on the network offering remote objects, then there will be also be many registries that the client may choose remote objects from.

The registry need only be accessed the first time that the remote object is defined on the client. To use the previous example, when `obj` is instantiated and made to point to `remoteObj`, the act of making it point to `remoteObj` involves a registry query. Subsequent method calls to `obj` are handled by the RMI runtime on the client, so the registry is only used for "bootstrapping" (i.e. setting up access to the remote object for the first time).

Because there can be more than one registry running on a host, and there may be many hosts serving remote objects on the network, there must be a mechanism for finding bound remote objects in the various registries. The LocateRegistry class contains static methods that return a reference to a registry on the current host, current host at specified port, a specified host or at a particular port on a specified host. Using this information, the bound remote objects in the registry can be listed, or a reference can be obtained for setting up a client object to access methods

in a remote object.

The diagram illustrates the process of setting up and accessing a remote object.

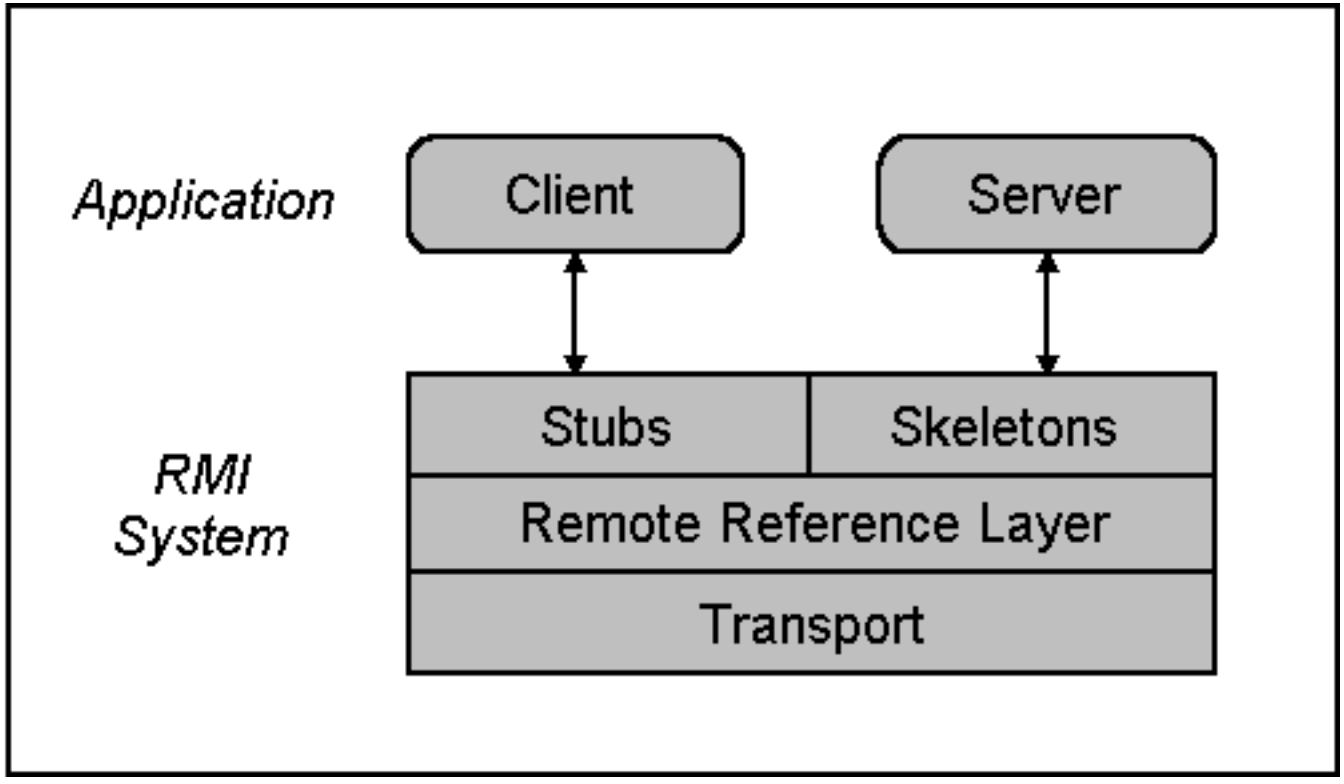


The RMI System

The RMI system consists of three layers:

- The stub/skeleton layer - client-side stubs (proxies) and server-side skeletons
- The remote reference layer - remote reference behavior (e.g. invocation to a single object or to a replicated object)
- The transport layer - connection set up and management and remote object tracking

The application layer sits on top of the RMI system. The relationship between the layers is shown in the following figure.



A remote method invocation from a client to a remote server object travels down through the layers of the RMI

system to the client-side transport, then up through the server-side transport to the server.

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote object as a conduit to the remote object. A client-held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote object and forwards invocation requests to that server object via the remote reference layer.

The remote reference layer is responsible for carrying out the semantics of the invocation. For example the remote reference layer is responsible for determining whether the server is a single object or is a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics-whether the server is a single object or is a replicated object requiring communications with multiple locations.

The transport is responsible for connection set-up, connection management, and keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport's address space.

In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to be done before handing off the request to the server-side skeleton. The skeleton for a remote object makes an up-call to the remote object implementation which carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer and transport on the server side, and then up through the transport, remote reference layer and stub on the client side.

In case you were wondering about the broadcast/multicast capability of a client, it is possible to implement point-to-multipoint communication strategies. The RMI package only includes UnicastRemoteObject, which is a class in the RMI package that is extended by the client interface object to allow point-to-point communication. If other strategies are desired, they would have to be developed.

REFERENCES

There's a brief explanation of how to locate remote objects in **http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-objmodel.doc.html#357**

See **http://chatsubo.javasoft.com/current/doc/rmi/rmiregistry.html** for a brief explanation of the remote registry script/batch file.

Some information about the remote reference layer was taken from:
http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-arch.doc.html

For information on how to get started with RMI (a quick tutorial), see:
http://chatsubo.javasoft.com/current/doc/tutorial/getstart.doc.html

The Java RMI Specification Table of Contents can be found at: **http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html**

Information about the LocateRegistry class can be found at: **http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-registry.doc.html#6378**

There is a brief explanation of how to locate remote objects in: **http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-objmodel.doc.html#357**