Amos Ng, Han Zhang

Design

The programmer realizes he needs to have remote method invocation capabilities in his program. That means he goes to Machine A to start up the:

RegistryServer

Keeps track of which hosts the remote objects are located on, and allows for binding of new remote objects. Backend class that application programmers do not interact with in code – but they must start this up whenever remote references are required.

Now the programmer goes to Machine B to create a local object X implementing:

Remote

An interface that all remote interfaces extend. Simply a marker for which classes can be accessed remotely and which cannot. Anything that is used as a parameter or a return value in a remote function must either implement this Remote interface or Java's serializable interface.

Now that X is created, the programmer wants X to be accessible remotely from other machines. He does this by getting an instance of the:

Registry

Connects to a running instance of the RegistryServer, and handles all queries and bind requests meant for it. Also invokes remote methods, listens for remote method invocations, and handles the creation of object stubs (either when requested explicitly, or when passing/returning objects by reference). Class that application programmers must interact directly with.

The programmer tells the Registry on Machine B that object X should be accessible remotely by anyone asking for "X String". The Registry on Machine B connects over the network to the RegistryServer on Machine A, and tells the RegistryServer that the object associated with "X String" resides on Machine B.

Now the programmer goes over to Machine C, where he wants to get a local version of object X. So he asks the Registry on Machine C for the object associated with "X String". The Registry on Machine C connects to the RegistryServer on Machine A to ask for the whereabouts of "X String", and the RegistryServer says it's on Machine B. So the Registry on Machine C returns to the programmer a:

Stub

Base stub class that all stub classes should inherit from. Stub classes are a simple local representation of a remote object. Stub functions simply tell the Registry which remote function of which remote object to invoke with which parameters. The Registry handles all the actual remote function invocation work. This is a backend class that application programmers are not supposed to interact with (at least when the stub compiler is created).

The Stub contains a:

Reference

Class that keeps track of which host and port a remote object is located on, as well as the String associated with that remote object.

So the application programmer now has an XStub on Machine C that contains a Reference to X on Machine B. The application programmer also has, on Machine C, a local Y object implementing the Remote interface, and a local Z object that implements the Serializable interface instead of the Remote interface.

The application programmer now calls XStub.doSomething(Y, Z). The stub's doSomething function gets the singleton instance of the local Registry on Machine C, and tells the Registry that it should call the function "doSomething" with the parameters Y and Z, on the object associated with "X string" on Machine B. The Registry creates a:

Message

A class encapsulating the name of the function to be invoked, the object it is to be invoked on, and the parameters to invoke the function with. The Message is completely hidden from the application programmer, and is only used by Registry to send and receive method invocation requests.

The Registry puts YStub inside the message, because Y implements Remote, so it will pass YStub by reference. The Registry puts Z inside the message, because Z implements Serializable instead of Remote, so Z will be passed by value instead. The strings "doSomething" and "X string" are also put inside the message. The message is now serialized and sent from the Registry on Machine C to the Registry on Machine B…

…and the Registry on Machine B receives the Message. It finds the object associated with "X string" – namely, X. It then finds the "doSomething" method of X that applies to objects of type Y and Z, and calls that function on the YStub and Z objects contained inside the Message. If "doSomething" calls any of the functions of YStub, the Registry on Machine B will have to invoke the corresponding function of Y on Machine C and wait for the result… but if "doSomething" calls or modifies anything of Z, only the local copy of Z on Machine B will be modified. In any case, "doSomething" eventually returns an object Y2 of the same class as Y. The Registry on Machine B creates a Y2Stub because Y2 also implements the Remote interface, so the Registry will be returning Y2 by reference. The Registry on Machine B serializes and sends the Y2Stub across the network to the Registry on Machine C…

…and the Registry on Machine C receives Y2Stub, and returns it to YStub as the result of the remote call on "doSomething". YStub then returns Y2Stub as the result of the local call to YStub.doSomething. Now the programmer has a Y2Stub to work with, but he does not care, and can continue calling functions on it as if it were a local object.

And all the objects lived happily ever after… except when a Stub throws a:

Amos Ng, Han Zhang

RemoteException

Base exception class for all errors caused by an error while invoking the function remotely. Perhaps there's a network communication error, or an object is not serializable nor remote… No matter what the problem is, this Exception is thrown when a remote function is invoked but does not return successfully.

What's finished

The ability to name remote objects

We're not sure whether this refers to the creation of object stubs or to the object string that an object is bound to on the RMI registry, or to the Reference class (which we don't have users interact directly with for simplicity – when you use Java's RMI you pretty much only have to interact with the Java Registry class as well).

No matter what this refers to, we're pretty damn sure we've got it covered.

The ability to invoke methods on remote objects

This works. Not only can you pass parameters by reference or by value, but results are also returned by reference or by value, all depending on whether something implements our Remote interface or not. If it doesn't, it must be Serializable so that it can be copied in its entirety over the network.

The ability to locate remote objects

We have a RegistryServer that works very well. What can I say? It works dude!

What's unfinished

A stub compiler

It should be very simple to create, as each function of the stub is composed of just one simple line. You can take a look at our handwritten stubs, ha.testing.simple.ComputationServerStub, ha.testing.zipcode.ZipCodeServerStub, and ha.testing.zipcode.ZipCodeRListStub to see the simplicity of our stubs for yourself. We just have to generate a serial version UID, the same constructor every time, and every function in the interface extending the Remote interface.

The automatic retrieval of stub .class files

Should be simple to create too – if the stub class to be instantiated is not found on the local machine, ask the remote one for the .class file, and the remote one will retrieve it and send it over the network. For now we have been testing where both the local and remote machines have the same set of defined classes.

A distributed garbage collection

Have clients tell the registry they're signing off when they quit. To take care of cases where they don't quit gracefully, ping clients every now and then, and remove their references when they don't respond.