Objective Viewpoint

Tutorials - 6.5
Mid-Summer 2000

Distributed Computation with Java Remote Method Invocation

by *Kevin Henry*

Spreading tasks among many workers can often accelerate their completion. For example, a single person, no matter how strong or fast, could never have built even the smallest of the Egyptian pyramids. However, by distributing the task among thousands of workers, the Egyptians were able to build these great wonders within a single lifetime.

Likewise, some computational problems, beyond the reach of all but the most powerful and expensive supercomputers, can be solved by a collection of modestly powerful and inexpensive "slave" computers. These slave computers are directed by a "master" coordinating computer. The idea of the master-slave relationship is central to a particular style of distributed computing.

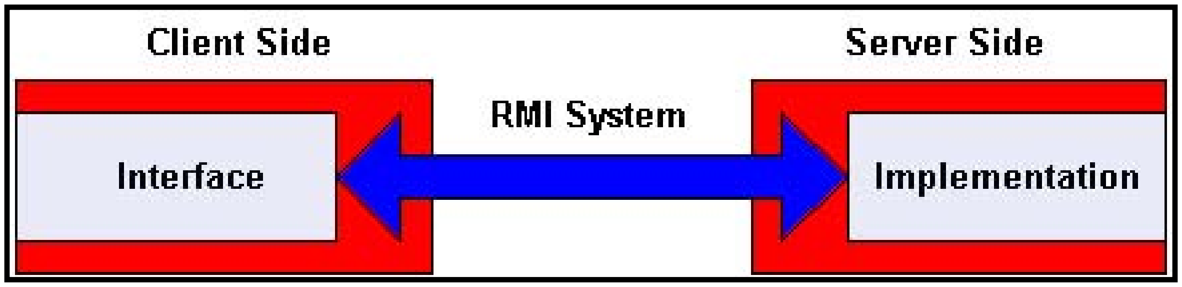
Distributed computing has become very popular in recent years for a few other reasons as well. Many complex problems have large data sets associated with them. These data sets are not easy to move from computer to computer. Nevertheless, it might be advantageous to have many computers working on the problem. A powerful server could control that data and distribute the parts of data its clients need. Also, a distributed system is much more fault tolerant than a single computer that brings the operation to a halt if it fails [6].

There are often semantic debates about the new terms **client** and **server**. Generally, a server provides a service that is used by a client. One reading would be that the programs doing the work *serve* their coordinating program. In the example to follow, we consider coordinating the efforts the service, performed by a server analogous to the master pyramid architect. The clients are the *slaves* that do the work. Try to understand this subtle and potentially confusing point as more of the example is explained.

Background

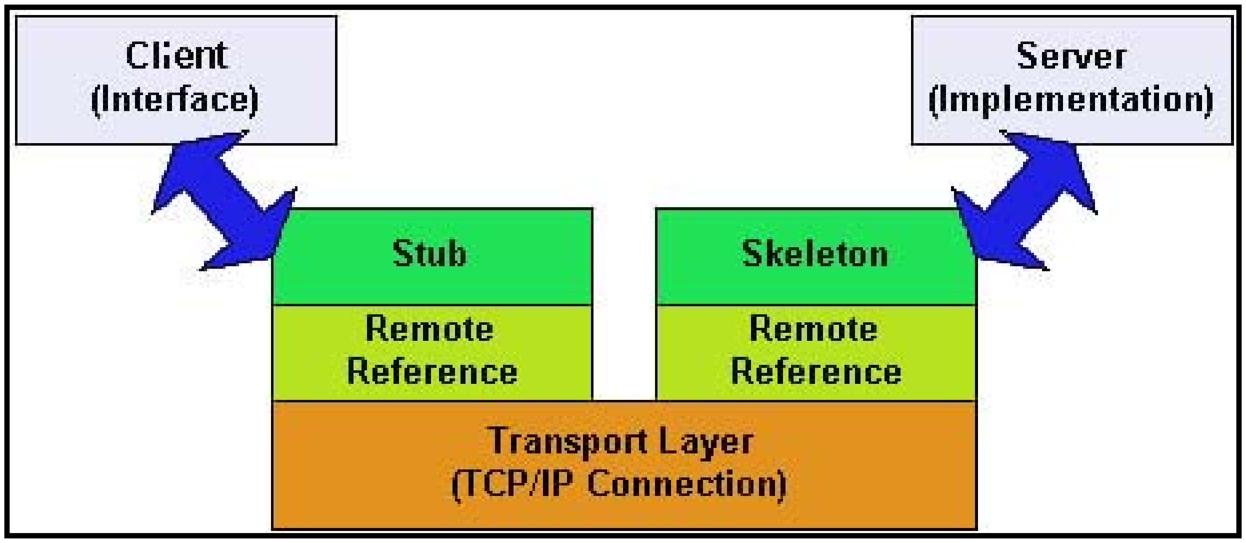
Java supports distributed computing in a variety of ways; one of the simplest is known as Remote

Method Invocation (RMI) [1]. Using the RMI capability of Java, clients may execute remote method calls on a server using a transparent interface. RMI provides a way for Java programs to interact with classes and objects working on different virtual machines on other networked computers. From the perspective of the client, objects on the server may be referenced as if they were local.



In the above diagram, the necessary components of a program using the RMI system are represented as a magical connection between a client and its server. A client is supplied with the interface of methods available from the remote server, but all implementation is left to the server side. In the truest sense, the details of implementing the remote methods are abstracted from the client software.

What really *connects* the client and server of an RMI system is a layered connection, transparent to developers of most simple applications, operated by the RMI subsystem of the two virtual machines involved.



The stubs and skeletons provide additional abstraction for the Java developer, once an RMI system is established. The code may be written as if all methods and objects are local to the client. The **remote reference layer** handles reference variables to remote objects, using the Transport layer's TCP/IP connection to the server.

What happens when a client tries to execute a method provided by the server? As mentioned above, the client has a stub of the method provided by the server, which executes the actual implementing class of that service. Any class whose objects are to be passed over RMI connections must be marked as such by declaring that they implement the `java.io.Serializable` interface. All arguments to a remote method must be either Java primitives or implement that interface.

A client requests a reference to an object from the server using the stub on the client side. The server gets the request from a skeleton on the server side. Between the two is the remote reference layer, which negotiates the requests by converting objects into portable form across the network. This conversion is called **marshalling**. If the data to be transferred is neither primitive nor serializable, the stub will throw a `MarshalException` [6]. The idea of serialization is mentioned again in the example to follow.

Example of a Distributed `KeyBlock` Checker

Complete and operable code for this demonstration is available from the author's website, along with instructions [5].

To distribute the work of a large problem, the server coordinates the work of the clients. Clients periodically retrieve their assignments from the server using remote method calls. For this example, I'll examine a distributed computing solution to a problem of cryptoanalysis. A message has been encrypted using a long integer key. For the sake of this example I assume can only be discovered by brute force checking all possible keys.

To implement the most efficient checking of all possible keys, a server will assign a range of unchecked keys to a client, which will check its assigned keys in parallel with all other clients. I want clients to request an assignment from the server in the form of a block of keys. I need a `KeyBlock` class, which must be marked as *serializable*.

```
public class KeyBlock
             implements java.io.Serializable {
   public int start, end;

   public KeyBlock (int s, int e)
   {
      start = s;
      end = e;
   }
}
```

The API for Remote Method Invocation is part of the standard Java Software Development Kit. With RMI, as will all other parts of the Java API (except `java.lang`) a class must be imported using an `import` declaration before it can be used in a program. If you need a class and don't know what package of the Java API it's in, try looking in the online documentation [2].

The server will assign blocks of keys to clients by returning an object from the `KeyBlockManager` class upon their request. A client side interface is used to inform clients what services are available and what to expect:

```
public interface KeyBlockManager
                 extends java.rmi.Remote {
   public KeyBlock getNext ()
                   throws java.rmi.RemoteException;
}
```

On the server side, we implement a class that is executed when the client makes a remote call. This class creates an instance of the `KeyBlock` class, and returns it.

```
public class KeyBlockManagerImpl
             extends java.rmi.server.UnicastRemoteObject
             implements KeyBlockManager {
   public KeyBlockManagerImpl ()
          throws java.rmi.RemoteException {
      super ();
   }

   public KeyBlock getNext ()
                   throws java.rmi.RemoteException {
      KeyBlock kb = new KeyBlock (nextStartIndex, BLOCK_SIZE);
      return kb;
   }
}
```

The developers of the Java remote method system provided a concept called the RMI registry, which runs on a generic port, and informs clients which port has the server to respond to their specific requests [3]. This provides an additional level of abstraction for developers of servers and clients. A registry provides a reference to a client looking for its server. The client code need not include the port where the server is running if it can look up that port dynamically from a registry. What port the registry uses is usually not such a difficult question to answer. Port 1099 is considered a default port for an RMI registry.

Using the RMI registry, a client can obtain a reference to an object that resides on a different computer (or maybe just a different virtual machine on the same computer) and call its methods as if that object were in the local virtual machine. Pay careful attention to the lines of code where the server code calls (`Naming.rebind`) to get a port assignment from the RMI registry. Notice also later that the client looks up the server (`Naming.lookup`) in the registry and then makes a request for an object reference. This process of using the `rmiregistry` is very similar to what the `portmap` daemon does on a Unix system to facilitate Remote Procedure Calls.

The server must coordinate requests for key block assignments from the clients. Assume that key blocks are a standard size and are assigned sequentially.

```
public class KeyBlockManagerServer {
   private static final String RMI_URL =
      "rmi://RMIRegistryServer:1099/KeyBlockManagerService";

   public KeyBlockManagerServer () {
      try {
         KeyBlockManager kbm = new KeyBlockManagerImpl ();
         Naming.rebind (RMI_URL, kbm);
      }
      catch (Exception e) {
         e.printStackTrace (System.out);
      }
   }

   public static void main (String [] args)
   {
      new KeyBlockManagerServer ();
   }
}
```

The client finds the server from the registry, and then gets a reference to a remote object that implements the `KeyBlockManager` interface from the RMI system. The client then enters a loop by executing a remote call to the remote object. The call to the remote object is handled identically to a call to a local object. The method call is transferred to the server, which executes the implementation and returns the assignment as a serialized object of the `KeyBlock` class.

```
public class KeyBlockClient {
   private static final String RMI_URL =
      "rmi://RMIRegistryServer/KeyBlockManagerService";

   public static void main (String [] args) {
      try {
         KeyBlockManager kbm =
            (KeyBlockManager) Naming.lookup (RMI_URL);
         while (true) {
               KeyBlock kb = kbm.getNext ();
               checkBlock (kb);
         }
      }
      catch (Exception e) {
         e.printStackTrace (System.err);
      }
   }
}
```

Running the RMI System

To actually make this RMI system operational, the source code files must be compiled into Java bytecode class files and distributed to the appropriate places. As a Java programmer you are already familiar with the `javac` tool to produce compiled bytecode class files. Part of the standard Java Software Development Kit (SDK) is another tool, `rmic`, which produces the stub and skeleton files necessary for the Remote Reference Layer mentioned earlier from the source code of the implementation [4]. The command works like this:

```
% javac KeyBlockManagerImpl.java
% rmic KeyBlockManagerImpl
```

Which produced the following new `.class` files:

`KeyBlockManagerImpl`	Compiled Java bytecode of the implementation, to be executed for a remote call.
`KeyBlockManagerImpl_Skel`	The skeleton of the object providing remote methods, used by the server side of the reference layer.
`KeyBlockManagerImpl_Stub`	The stub of the object providing remote methods, used with the interface by the client side of the reference layer.

The respective class loaders must also have class files for the following available:

Server Side	**Client Side**
Remote service interfaces	Remote service interface definitions
Remote service implementations	Stubs for the remote implementations
Skeletons for the implemented classes	Any values returned by remote calls
Stubs for the implemented classes	Any other classes the client needs

Any other classes the server needs	

Note especially that clients must have available the class definition of any return type of a remote method. In our primitive example, the client's class loader must find `KeyBlock`. To start an RMI registry and bind the server to it:

```
% rmiregistry &
% java KeyBlockManagerServer &
```

```
This prepares two Java virtual machines to handle requests for remote
objects from clients.
```

```
% java KeyBlockClient
```

```
The client makes a request to the registry to find the appropriate
server. Then it makes a request to its server for a reference to an
object that represents its task.
```

What's Next?

There is another new technology emerging that may change the face of cross platform distributed and client-server computing. Developers are now working closely to make the processing much less Java-specific. One idea that has much potential is Common Object Request Broker Architecture (CORBA). As the name implies, CORBA is a standard for location transparency and language independence between creators and users of objects. A client would neither know nor care about where an object was or in what language it was implemented [7].

The way RMI is implemented now, with serialized object streams, is inherently too slow. You will have noticed a speed problem even in the trivial example above. A service distributed with CORBA will be faster and more universal than a similar service distributed with RMI. The main reason for the performance increase will be the Internet InterOperability Protocol (IIOP), which quickly creates very high speed connections between objects while minimizing overhead [7].

Conclusion

Hopefully you now have a glimpse of the power of distributed computing with remote calls, and will consider implementing solutions in that way. If you wish to download the full source code of a functional version of this distributed application, visit the author's web site [5]. Feel free to experiment with it. Enjoy!

References

1

Sun Microsystems, Java Remote Method Invocation -- http://java.sun.com/products/jdk/rmi/

2

Sun Microsystems, Java API Documentation -- http://java.sun.com/j2se/1.3/docs/api/

3

Sun Microsystems, `rmiregistry` Tool Documentation -- http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/rmiregistry.html

4

Sun Microsystems, `rmic` Tool Documentation -- http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/rmic.html

5

Kevin Henry, A Trivial Distributed RMI Application -- http://www.ece.vill.edu/~khenry/rmiapp/

6

Flanagan, Farley, Crawford, Magnusson, *Java Enterprise in a Nutshell* -- published by O'Reilly & Associates

7

Hughes, Shoffner, Hamner, *Java Network Programming*, 2nd Ed. -- published by Manning Publications Co.