AOS ASSIGNMENT

Programming Project

Peer to Peer File Sharing System Using Java RMI

Abstract

Java Remote Method Invocation (RMI) is a built-in and easy-to-use framework for the distribution of remote Java objects. Its simplicity and seamless inter-virtual machine communication has made it a valuable tool for distributed services. Here we present an approach that makes Java RMI usable for P2P and similar distribution models. The solution basically consists of following ideas: (1) registering the clients to the central indexing server, (2) searching for a file and locating the corresponding peer containing it, and (3) downloading the file from one peer to another peer, (4) calculating the average response time it takes to make 1000 sequential requests, and (5) calculating response times when multiple clients are concurrently making requests to the indexing server

Java RMI:

Remote Method Invocation Java RMI is a mechanism to allow the invocation of methods that reside on different Java Virtual Machines (JVMs). The JVMs may be on different machines or they could be on the same machine. In either case, the method runs in a different address space than the calling process.

Participating Processes:

There are three entities involved in running a program that uses RMI: client: this is the program that you write to access remote methods server: this is the program that you write to implement the remote methods - clients connect to the server and request that a method be executed. The remote methods to the client are local methods to the server. Object registry: this is a program that you use. The object registry runs on a known port (1099 by default) A server, upon starting, registers its objects with a textual name with the object registry. A client, before performing invoking a remote method, must first contact the object registry to obtain access to the remote object.

The implementation has the following steps:

1. **Defining the remote interface**

A remote object is an instance of a class that implements a *remote interface*. A remote interface extends the interface java.rmi.Remote and declares a set of *remote methods*. Each *remote method* must declare java.rmi.RemoteException (or a superclass of RemoteException) in its throws clause, in addition to any application-specific exceptions.

2. **Implemet the server**

A "server" class, in this context, is the class which has a main method that creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name in a Java RMI *registry*. The class that contains this main method could be the implementation class itself, or another class entirely.

In this program, the main method for the server is defined in the class ThreadOfHelloServer. The server's main method does the following:

- Create and export a remote object: The main method of the server needs to create the remote object that provides the service
- Register the remote object with a Java RMI registry: For a caller (client, peer, or applet) to be able to invoke a method on a remote object, that caller must first obtain a stub for the remote object. A Java RMI registry is a simplified name service that allows clients to get a reference to a remote object. In general, a registry is used (if at all) only to locate the first remote object a client needs to use. Then, typically, that first object would in turn provide application-specific support for finding other objects. For example, the reference can be obtained as a parameter to, or a return value from, another remote method call.
- The static method LocateRegistry.getRegistry that takes no arguments returns a stub that implements the remote interface java.rmi.registry.Registry and sends invocations to the registry on server's local host on the default registry port of 1099. The bind method is then invoked on the registry stub in order to bind the remote object's stub to the name "Hello" in the registry.

The implementation class Hello implements the remote interface HelloInterface, providing an implementation for the remote method registerFiles and search.

3. Implement the peers

The client program obtains a stub for the registry on the server's host, looks up the remote object's stub by name in the registry, and then invokes the registerFiles method on the remote object using the stub.

This client first obtains the stub for the registry by invoking the static LocateRegistry.getRegistry method with the hostname specified on the command line. If no hostname is specified, then null is used as the hostname indicating that the local host address should be used.

Next, the client invokes the remote method lookup on the registry stub to obtain the stub for the remote object from the server's registry.

Finally, the client invokes the registerFiles method on the remote object's stub, which causes the following actions to happen:

- The client-side runtime opens a connection to the server using the host and port information in the remote object's stub and then serializes the call data.
- The server-side runtime accepts the incoming call, dispatches the call to the remote object, and serializes the result to the client.
- The client-side runtime receives, deserializes, and returns the result to the caller.

The response message returned from the remote invocation on the remote object is then printed to System.out.

4. Compile the source files

The source files for this example can be compiled as follows:

```
javac -d RMI_Server ThreadOfHelloServer.java ThraedOfHelloClient.java Hello.java
HelloInterface.java HelloClient.java FileDetails.java FileImpl.java AverageResonseTime.java ConcurrentSearchTime.java
```

5. **Start the Server and Run the clients**

Start the server:

```
java -classpath RMI_Server ThreadOfHelloServer
```

Run the clients:

```
java  -classpath RMI_Server ThreadOfHelloClient
```

6. **Compute the average response time per client search request**
   In the class AverageResponseTime.java , a peer connects to another peer in the network in order to download the file from it. The file to be downloaded method “DownloadFromPeer” is called sequentially for 1000 times. The average response time for this process is calculated and displayed.

7. **Measure the response times when multiple clients are concurrently making requests to the indexing server.**
   In the class ConcurrentFileSearch.java,the time taken for a single peer to download a file from a specified peer is computed. Consecutively the time taken to download a file when two or more clients are simultaneously downloading a file from the specified same peer acting as a server is also computed. Here we can compare the time taken among different peers when running concurrently, to conclude on the performance.

Improvements to the RMI Programming

Frequently, you can simply rework the relevant APIs to speed things up and then, at that point, stop. If not, you must optimize the serialization step of your RMI calls.

Though serialization provides great flexibility, its overhead can be quite high. This is because serialization uses Java's reflection mechanism to determine what data to serialize and how to serialize it. One can convert objects to byte arrays more efficiently by writing out objects yourself with Java's externalization mechanism.