Remote Data Analysis Using Java

Stefan Kuhlins and Martin Schader
Lehrstuhl für Wirtschaftsinformatik III
Universität Mannheim
D–68131 Mannheim
Germany

Abstract

In this paper we examine the use of Java's networking capabilities in a client/server application for data analysis. On the client side runs an applet which collects analysis requests. The applet sends the requests to the server where the analysis actually takes place. The server sends the analysis results back to the client which displays them graphically. We demonstrate the use of Java's RMI by means of a simple classification example.

1 Motivation

In a typical scenario there are large amounts of data as well as algorithms for data analysis at one place and remote requests for data analysis over the Internet should be possible. This scenario could be implemented as a client/server application. In doing so the data is in a protected environment on the server where data is condensed and only the analysis results are sent back over the net to clients. Now, in comparison to sending the original data less resources are needed. On the other hand, a powerful server is required for data analysis while representation of results on the client side demands only limited computational power. Figure 1 shows the general structure of such a connection.

2 An example

Since we want to concentrate on client/server programming techniques using Java we chose a very simple example for the server side.

The aim is to cluster a set of n objects that are characterized by m variables via the k-means algorithm (Hartigan & Wong 1979). In order to be able to easily represent clustering results, we restricted the data type of each variable to $\mathbb{R}$ and

*Wolfgang Gaul und Hermann Locarek-Junge (eds.): Classification in the Information Age, Proceedings of the 22nd Annual Conference of the Gesellschaft für Klassifikation e.V., 4–6. March 1998 in Dresden, Springer-Verlag, 1999, pp. 359–367

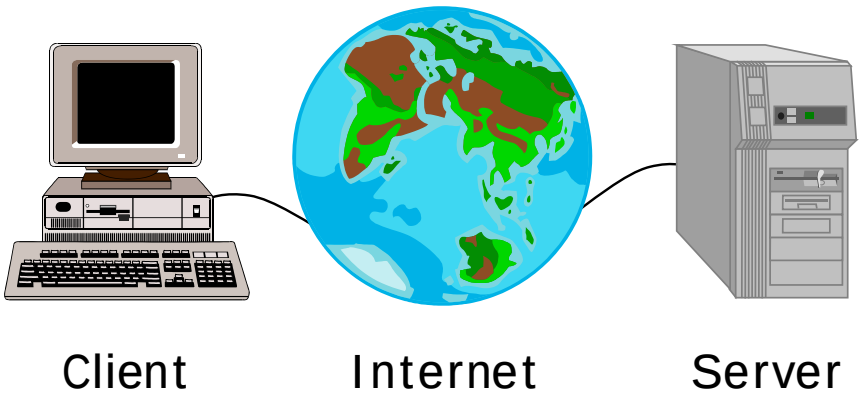

Figure 1: Connection of client and server over the Internet

set $m = 2$. (Of course, we could also have chosen an algorithm that processes qualitative or mixed data and produces results in $\mathbb{R}^2$.) A user of the system has to select one of the data sets available on the server and must provide the maximal number k of clusters to be computed.

3 High level overview

The first step on the client side is to start a browser and load the HTML page with the applet from the server. You need to know the URL of that page, for example:

http://www.wifo.uni-mannheim.de/dresden98/client.html

Next, you select the desired data set as well as the number of clusters. The request is then sent to the server and the client waits for the answer. The server program computes the result and sends it back to the client. Finally, the client presents the corresponding graphic and waits for new input.

After the server is started it waits for client requests. If such a request occurs the server loads data from the local database analyzes it and sends the results back to the client.

4 Implementation techniques

By now, numerous implementation techniques exist for interactive HTML-pages. A rather new and promising technique is the use of Java on the client side as well as on the server side. In this paper we demonstrate how the interaction between client and server can be achieved with RMI (*Remote Method Invocation*). Besides RMI you can use low level sockets, RPC (*Remote Procedure Call*), and CORBA (*Common Object Request Broker Architecture*). In view of our experiences, RMI is the clear winner for Java-to-Java interprocess communication. Some references

concerning RMI that we can recommend are (Berg & Fritzinger 1998), (Orfali & Harkey 1998), (Sun Microsystems Inc. 1998), and (Vanderburg 1997).

For the client side we implemented an applet, which lets the user specify the input data (filename and number of clusters) for the analysis algorithm. Since an applet can only establish connections to the server it came from the user cannot change the server name. Likewise, the port is not editable because we solely use the default port. The HTML-page which embeds the applet looks like this:

```
<html>
<head>
<title>Cluster Analysis</title>
</head>
<body>
<center>
<h1>Cluster Analysis</h1>
<applet code="Client" width="650" height="500">
This browser doesn't know the APPLET-tag.
</applet>
</center>
</body>
</html>
```

After a successful cluster analysis the browser shows a graphic like the one in figure 2.

5 Remote Method Invocation (RMI)

Objects communicate through messages, i.e. one object calls a method of another object. In our example, the client object has to call a server method which actually performs the cluster analysis. The problem here is that client and server objects reside on different computers and therefore in different Java virtual machines. They have to communicate over a network.

Java offers the RMI package for interprocess communication between Java virtual machines. After some preparation RMI enables a method of an object in one virtual machine to call a method of an object in another virtual machine with the same syntax and ease as a local method invocation.

On the basis of a simple example for a cluster analysis we want to explain and demonstrate how the capabilities of RMI are used in practice. In doing so we will concentrate on the code snippets which are relevant for RMI.

6 Implementation of the server side

Methods that should be callable remotely have to be declared in an interface which extends java.rmi.Remote. Only those methods specified in a remote interface are

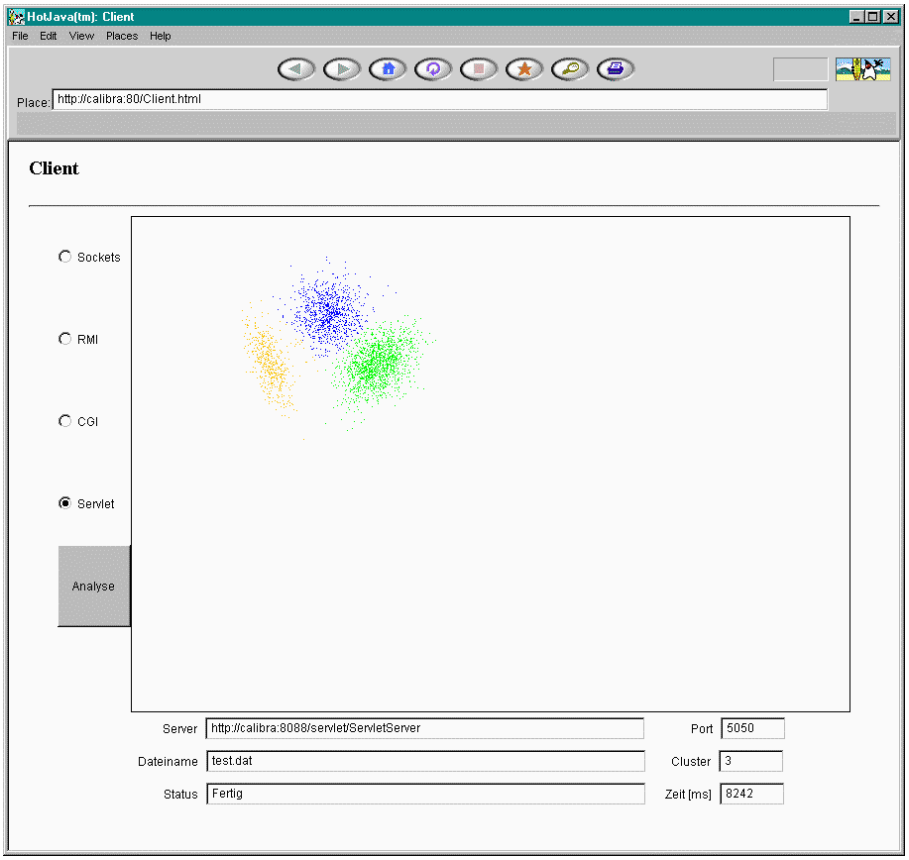

Figure 2: Screenshot of the client side

available remotely. Such methods must have java.rmi.RemoteException declared in their throws clause. A remote method invocation across a network can cause numerous errors. This exception provides a mechanism to gracefully handle unlikely but possible failure scenarios. One possible interface declaration for our remote object would be the following:

```
// RMIServerInterface.java

import java.rmi.*;

interface RMIServerInterface extends Remote {
   public ClusterAnalysis kmeans(String filename,
      int clusterCount) throws RemoteException;
}
```

Next, we define the kmeans method in the RMIServer class which implements our RMIServerInterface. The RMIServer class extends java.rmi.server.Unicast-

RemoteObject. This class provides support for point-to-point active object references (invocations, parameters, and results) using TCP streams. By extending UnicastRemoteObject, RMIServer will be automatically "exported" and is ready to be used outside the virtual machine in which it was created.

```
// RMIServer.java

import java.rmi.*;
import java.rmi.server.*;

class RMIServer
      extends UnicastRemoteObject
      implements RMIServerInterface {

   public ClusterAnalysis kmeans(String filename,
         int clusterCount) throws RemoteException {
      try {
         FileInputStream fis
            = new FileInputStream(filename);
         ObjectInputStream ois
            = new ObjectInputStream(fis);
         Data d = (Data) ois.readObject();
         ClusterAnalysis ca
            = new ClusterAnalysis(clusterCount, d);
         ca.classify();
         return ca;
      }
      catch (Exception e) {
         ...
         return null;
      }
   }
   ...
}
```

The kmeans method reads the data file, classifies the data, and returns the result. The arguments and return values of remote methods must implement the Serializable interface. So does the ClusterAnalysis class.

```
class ClusterAnalysis implements Serializable {
   ...
}
```

The main method of the server class sets an RMI security manager. If no security manager has been set, RMI will only load classes from local system files

as defined by the environment variable CLASSPATH or a path provided with the -classpath option.

```
public static void main(String[] args) {
   try {
      System.setSecurityManager(
         new RMISecurityManager());
      Naming.rebind("RMIServer", new RMIServer());
   }
   catch (Exception e) { ... }
}
```

The next step is to create an RMIServer object and bind it to a unique name. A remote object can be bound to any name, but you have to be aware of name collisions.

Even if the body of the constructor is empty you have to define a default constructor because the constructor of the superclass UnicastRemoteObject can throw a RemoteException.

```
public RMIServer() throws RemoteException { }
```

The server implementation itself is finished now, but there is some more work to do before clients can connect.

7 Implementation of the client side

For the client side of our client/server application we implement an applet. In the following we will concentrate on the code for the communication with the server.

```
// Client.java

public class Client extends Applet
   implements ActionListener {
   ...
}
```

When a browser or applet viewer loads an applet the init method is called to inform the applet that it has been loaded into the system and can perform its initialization. Here, we first set an RMI security manager so that we can connect to the remote server. Additionally, init has to provide code for the user interface shown in figure 2.

```
public void init() {
   if (System.getSecurityManager() == null)
```

```
        System.setSecurityManager(new RMISecurityManager());
      ...
}
```

After the user has pressed the Analysis button the actionPerformed method is called. Within this method body the client applet connects to the server.

```
public void actionPerformed(ActionEvent e) {
      ...
      try {
         RMIServerInterface si = (RMIServerInterface)
            Naming.lookup("//" + server + "/RMIServer");
         ClusterAnalysis ca
            = si.kmeans(filename, clusterCount);
      }
      catch(Exception x) { ... }
      ...
}
```

The lookup method returns the remote object for the given URL. An RMI URL looks very much like any other URL. It has the general form

rmi://host:port/name

where host is the host name of the registry. It defaults to the current host. Applets can retrieve a reference to a remote object only from the server from which the applet came. port specifies the port number of the registry and defaults to the registry port number 1099. name identifies the remote object on the server.

With the remote object at hand we are able to call the method kmeans which performs the cluster analysis on the server. It looks like any ordinary method call, but it really goes through a stub (see figure 3).

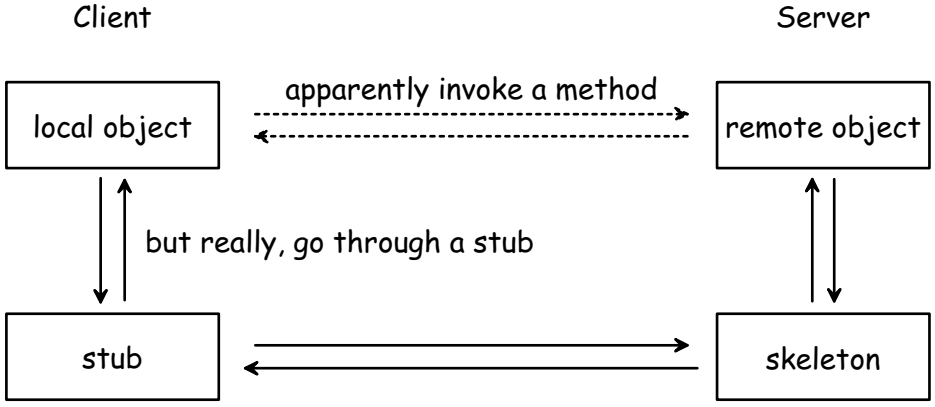

Figure 3: Stubs and Skeletons

8 Stubs and skeletons

A *stub* is a client-side proxy that implements the remote methods of a remote object. The stub is responsible for packaging (serializing) the arguments of a remote method invocation and passing control to the server. A *skeleton* is the corresponding server-side proxy that accepts a method invocation from a client, unpacks any arguments and dispatches the invocation to the target method on the server.

Stub and skeleton class files are created for the server object implementing the java.rmi.Remote interface by the rmic compiler. In our example the command

```
rmic RMIServer
```

produces the files RMIServer_Stub.class and RMIServer_Skel.class. Both files must be available to the virtual machine that is exporting the remote object. The clients of a remote object need access only to the stub class file. You can copy the stub file to the client or make it available from an URL.

9 RMI registry

The rmiregistry command creates a remote object registry on a specific port. On a Windows box we do this by:

```
start /min rmiregistry
```

It is possible to specify a port number. If the port number is omitted, it defaults to 1099. The remote object registry is a bootstrap naming service which is used by RMI servers.

10 Starting the server

Finally, the server program itself must be started:

```
start java RMIServer
```

Now the server is ready to answer client requests like the one displayed in figure 2.

11 Summary

The RMIServer class creates and exports a remote object. The Client class looks up the remote object in a registry and calls the kmeans method defined for the remote object. RMIServer defines the implementation for the remote object. A ClusterAnalysis object is created on the server and a copy is sent back to the client. The communication dynamics between client and server are shown in figure 4.

Because this paper is all about RMI and not about good programming practices, we show how to compile and run the application from a single directory. In practice, you would probably want to keep your source code outside your Web server's directory tree.

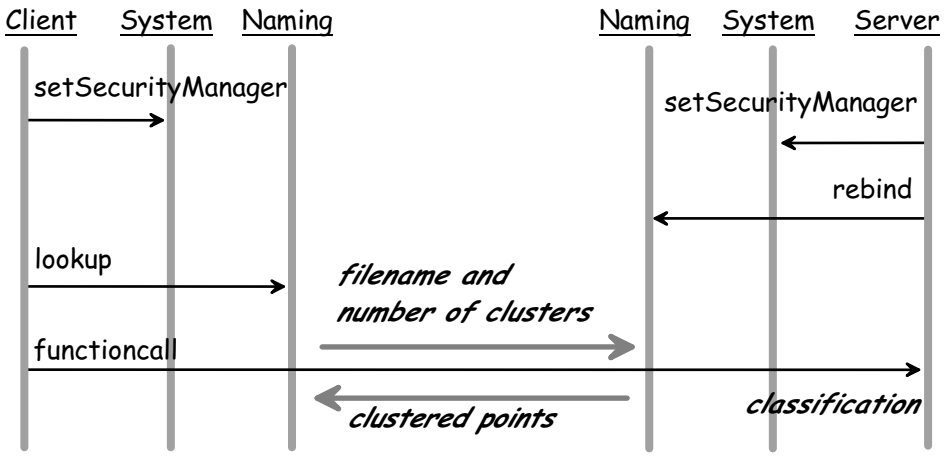

Figure 4: Communication between client applet and server

12 Conclusion

Java simplifies the task of client/server programming. RMI is the right technique for Java-to-Java interprocess communication. It enables a method of an object in one virtual machine to call a method of an object in another virtual machine with the same syntax and ease as a local method invocation. Applications like remote data analysis are good examples of Java's potential.

References

Berg, D. & Fritzinger, J. (1998), *Advanced Techniques for Java Developers*, Wiley Computer Publishing.

Hartigan, J. A. & Wong, M. A. (1979), 'Algorithm AS136: A k-means clustering algorithm', *Applied Statistics* **28**.

Orfali, R. & Harkey, D. (1998), *Client/Server Programming with Java and CORBA*, 2nd edn, John Wiley & Sons, New York.

Sun Microsystems Inc. (1998), 'RMI – remote method invocation'. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/.

Vanderburg, G. (1997), *Maximum Java 1.1*, Sams.net Publishing.