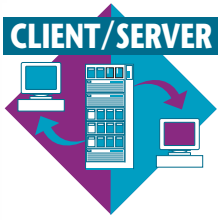
Developing a Java Client/Server Prototype



By Dr. Peter W. Farrett

This article outlines general steps for developing a Java-based client/server application. The client and server discussions highlight important concepts and techniques that are fundamental for developing Java™ client/server applications. The article also includes complete source code for the application described.

Client/server applications generally consist of processing queries where the client utilizes services provided by the server. Communication must be established and no loss of data can occur.

In the client/server application described in this article, the client maintains the user information (member data) and the server handles the requests for information. Specifically, the application will read from the client and write data to a user member file via the server with appropriate server system messages. The server will also send user information back to the client.

Every Java client/server application, including the one described in this article, contains several important components:

- **Graphical user interface (GUI):** A Java component that is implemented with a subclass of the Abstract Windowing Toolkit (AWT) component class. In our example, the client has a simple GUI consisting of a text field, string, and several buttons and labels; the server side does not require a GUI.
- **Input and output streams:** Java file I/O. These streams must read both the client and server data. For this application, we used the Java package, `java.io`, which contains input and output streams (read/write). The `InputStream` and `OutputStream` classes (in `java.io`) are abstract superclasses that define I/O behavior.
- **Sockets:** Both the client and server programs require network communication. These low-level methods "bind" each other via a communication link—from endpoint to endpoint. The `java.net` package implements the client- and server-side with `Socket` and `ServerSocket` classes, respectively.
- **Multithreading:** A thread is a single control flow (execution) within a program. In our application, we implement the `start()`, `stop()`, and `destroy()` methods relative to our threads of execution since numerous, simultaneous threads will occur based on client requests.

The Client (Applet)

The client must read and write input data. The `loadMember()` and `writeMember()`

Peter W. Farrett

methods handle this requirement along with socket connection objects. The client must first obtain the server's connection. For example, `writeMember()` method is accomplished by the `socket` object, a string that identifies the hostname, and port address, which creates a new socket. Figure 1 shows how the client creates a new socket.

```
socket s = null;
theServer = "austin.ibmtest.com";
thePort = 8081;

s = new Socket(theServer,thePort);
```

Figure 1. Creating a new socket

In order to read member data from the server's file system, the client must access Java I/O streams that relate to the socket connection. This "connects" `socket s` with the Java `input` stream. The subsequent lines of code wait for the server to read a file, then send back user information, as shown in Figure 2.

Conversely, Figure 3 shows how to write data from the client (to be written at the server's location via `ServerSocket`) and send the request to the server for processing.

Once the socket is connected with the `getOutputStream()` method, user information (`companyname`, `addr1`) is sent out through this pipeline. The `flush()` method follows each `println()`, because flushing is necessary after each statement. Finally, file termination (end of transmission) denotes an end of file (EOF) delimiter.

The remaining code implements other Java features, such as error handling, the proper closing of I/O streams, and event-driven actions. The main function of this client example is to request user information from the server, send user information to the server, and terminate the transaction at completion. See complete source code at the end of this article.

The Server (Application)

One fundamental task of a server is to accommodate requests from multiple clients without halting, blocking, or stopping a client request. This involves two important concepts:

- The Java `ServerSocket` object (in `java.net`) is used so the server listens to a specific port.

```
DataInputStream in = new DataInputStream(s.getInputStream());

line = in.readLine().trim();
while(!line.equals("end of transmission"));
{
  text.appendText(line + "\n");
  System.out.println(line);
  line=in.readLine().trim();
}
```

Figure 2. Reading data from the server

```
PrintStream out = new PrintStream(s.getOutputStream());
out.println(companyname);
out.flush();
out.println(addr1);
out.flush();
out.println("end of transmission");
out.flush();
```

Figure 3. Writing data to the server

```
ServerSocket s Socket;
...
...
while(running) {
  Socket soc = null;
  try {
    soc = sSocket.accept();
    serverRequest(soc);
  }
  catch(IOException error){}
}
```

Figure 4. Requests from multiple clients

- The server multithreads all client requests, as shown in Figure 4. A simple loop handles client-side requests.

The `serverRequest()` call creates a new server object and immediately starts a single thread for each client request and invokes the Java `run()` method. Similar (client) coding techniques are used with respect to our previous use of I/O streams, `println()` and `flush()` methods; however, file I/O is added for reading and writing member information to a file, as shown in Figure 5.

It is important to remember to close all files and their associated streams in the order in which they occurred after I/O and file processing. Figure 6 shows an example. Also, error handling should be implemented when appropriate.

Finally, Figure 7 shows how we could override the Java `stop()` method with our own `stopThread()`.

```
in.close();   //file #1 for read
out.close();  //stream associated with file #1
out2.close(); //file #2 for read/write
in2.close();  //stream associated with file #2
```

Figure 6. Closing files and associated streams

```
public void stopThread(Thread ssThread)
{
  if (ssThread != null)
  {
  ssThread.stop();
  ssThread.destroy();
  }
}
```

Figure 7. Overriding Java's stop() method

Invoking a `stop()` and `destroy()` method accomplishes two things:

- It ensures that the running thread has stopped, based on a client transaction.
- It frees system resources. See the complete source code at the end of this article.

Complete Source Code

The following sections provide complete source code for the Java client/server application.

Figure 8 shows the client code, Figure 9 shows the server code, and Figure 10 shows the HTML code.

```
File file = new File("memberinfo.txt");
RandomAccessFile in = new
                      RandomAccessFile(s.getOutputStream(file,"r"));
PrintStream out = new PrintStream(s.getOutputStream());

DataInputStream in2 = new DataInputStream(new BufferedInputStream
                          (s.getInputStream()));
File file2 = new File("member" + counter + ".txt");
RandomAccessFile out2 = new RandomAccessFile(file2,"rw");
```

Figure 5. Reading and writing member I/O

```
import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;


public class ProtoClient extends Applet
{
  String theServer;
  int thePort;
  Label title = new Label("SDP Registration Form", Label.CENTER);
  Label inputField1 = new Label("Name");
  Label inputField2 = new Label("Address");
  TextArea text = new TextArea();
  Button submit = new Button("Submit New Member Info");
  String companyName, addr1;

  TextField companyname = new TextField("",10);
  TextField addr1 = new TextField("",10);

  public void init()
  {

        add(title);
        add(text);
        add(inputField1);
        add(companyname);
        add(inputField2);
        add(addr1);
        add(submit);

addNotify();
resize(411,322);
  }


void loadMember()
{
Socket s = null;
theServer = "austin.ibmtest.com";
thePort = 8081;

try {
    s = new Socket(theServer, thePort);
System.out.println ("client found, waiting for a connection...") ;
java.io.InputStreamReader(s.getInputStream())) ;
DataInputStream in  = new DataInputStream(s.getInputStream());
String line = in.readLine() ;
text.appendText(line + "\n") ;
System.out.println("connected") ;

int counter = 0 ;

//Wait for server to read a file and send each line
line = in.readLine().trim() ;
          while(!line.equals("end of transmission"))
          {
```

(continued on following page)

Figure 8. Client code

(continued from previous page)

```
            text.appendText(line + "\n");
            System.out.println(line) ;
            System.out.println(counter++) ;
            line = in.readLine().trim() ;
           }

System.out.println("sleeping for 2 seconds...") ;
try
{
Thread.sleep(2000) ;
}
catch(InterruptedException e2)
{
    System.out.println(e2.toString()) ;
}

System.out.println("waking up...") ;
java.io.OutputStreamWriter(s.getOutputStream())) ;
PrintStream out = new PrintStream(s.getOutputStream());
out.println("end of transmission") ;
out.flush() ;
System.out.println("finished writing data") ;

     }
     catch (IOException e) {
        showStatus("Server not activated: " + e);
     }
     finally {
        if (s!=null) try { s.close(); } catch (IOException e) {}
     }
  }


  void writeMember()
  {
     Socket s = null;
     theServer = "austin.ibmtest.com";
     thePort = 8081;

     CompanyName = companyname.getText();
     Addr1 = addr1.getText();

     try {
       s = new Socket(theServer, thePort);
System.out.println ("client found, waiting for a connection...") ;
java.io.InputStreamReader(s.getInputStream())) ;
DataInputStream in  = new DataInputStream(s.getInputStream());
String line = in.readLine() ;
text.appendText(line + "\n") ;
System.out.println("connected") ;

int counter = 0 ;

//Wait for server to read a file and send each line
line = in.readLine().trim() ;
```

(continued on following page)

Figure 8. Client code

(continued from previous page)

```
        while(!line.equals("end of transmission"))
{

       text.appendText(line + "\n");
System.out.println(line) ;
line = in.readLine().trim() ;
}

System.out.println("sleeping for 2 seconds...") ;
try
{
Thread.sleep(2000) ;
}
catch(InterruptedException e2)
{
System.out.println(e2.toString()) ;
}

System.out.println("waking up...") ;
java.io.OutputStreamWriter(s.getOutputStream())) ;
PrintStream out = new PrintStream(s.getOutputStream());
out.println(companyname) ;
out.flush() ;
out.println(addr1) ;
out.flush() ;
out.println("end of transmission") ;
out.flush() ;
System.out.println("finished writing data") ;

     }
     catch (IOException e) {
        showStatus("Server not activated: " + e);
     }
     finally {
        if (s!=null) try { s.close(); } catch (IOException e) {}
     }



  }


  public boolean action(Event e, Object o)
  {
     if(e.target == submit) {
      System.out.println("I'm in submit") ;
        writeMember();
        System.out.println("I just submitted") ;
        return true;
     }
    return false;
  }


}
```

Figure 8. Client code

```
import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;

public class ProtoServer extends Thread
{
  static final int defPort = 8081;
  static Socket s;
  static int counter = 0;
  boolean tFinished;
  static Thread sThread;

static void serverRequest(Socket ss)
{
boolean tFinished = false;
counter = counter + 1;
s=ss;
Thread thread;
ProtoServer sServer = new ProtoServer();
thread = new Thread(sServer);
sThread = thread;
thread.start();

}

public void run()
{
try
{
String line,line2;
File file = new File("stub.txt") ;
RandomAccessFile in = new RandomAccessFile(file, "r") ;
java.io.OutputStreamWriter(s.getOutputStream())) ;
PrintStream out = new PrintStream(s.getOutputStream()) ;
//Signal that a connection has been established.
System.out.println("connection established") ;
//Read a file and send text to client
while(in.getFilePointer() < in.length())
{
out.println(in.readLine()) ;
out.flush() ;
}
out.println("end of transmission") ;
out.flush() ;

//Get text from client and append to a file
DataInputStream in2 = new DataInputStream(new
BufferedInputStream(s.getInputStream())) ;
File file2 = new File("mem" + counter + ".txt") ;
RandomAccessFile out2 = new RandomAccessFile(file2, "rw") ;
System.out.println("about to enter second while loop to write a file")  ;
line2 = in2.readLine().trim();
while(!line2.equals("end of transmission"))
{
System.out.println(line2) ;
out2.writeBytes(line2 + "\n") ;
line2 = in2.readLine().trim();
}


in.close() ;
out.close() ;
out2.close() ;
```

(continued on following page)

Figure 9. Server code

(continued from previous page)

```
in2.close() ;
System.out.println("files closed");
//stop thread from running and free up resources
stopThread(sThread);

} catch(IOException w) {}

}

public void main(String args())
{
boolean running = true;
int thePort;
ServerSocket sSocket;

if (args.length == 1)
    thePort = new Integer(args()).intValue();
else
    thePort = defPort;

System.out.println(thePort) ;

     try {
       sSocket = new ServerSocket(thePort);
     }
       catch (IOException e) {
       System.err.println("I/O Server exception 1: " + error);
       return;
     }

     System.err.println("listening on " + thePort);
     while(running) {
       Socket soc = null;
       try {
       soc = sSocket.accept();

     System.out.println("it is accepted") ;
     System.out.println("about to call serverRequest") ;
     serverRequest(soc);
     System.out.println("handled the request") ;
       }
     catch(IOException e) {
       System.err.println("I/O Server exception 2: " + error);
       }

     }
   }

   public void stopThread(Thread ssThread)
   {
     if (ssThread != null)
       {
         ssThread.stop();
         ssThread.destroy();
       }

   }



}
```

Figure 9. Server code

```
<HTML>
<BODY>
<APPLET
  CODE=ProtoClient.class
  WIDTH=400
  HEIGHT=200>
</APPLET>
</BODY>
</HTML>
```

Figure 10. HTML code

Figure 11 shows the client registration form.

Figure 11. Registration form from client

Conclusion

This article discusses a stand-alone server and a client applet portion of a Java client/server prototype. Fundamental client/server concepts and techniques are illustrated including file I/O, client and server connections, port requests, reading/writing to sockets, "cleaning-up" by closing all I/O streams, and handling client-side transactions via multithreading. It is easy to quickly prototype "compact" Web applications because Java provides easy socket programming via `Socket` and `ServerSocket` classes. It is also very functional regarding input and output data streams.

Acknowledgments

The author would like to thank Mickey Nix for technical assistance with helping to debug part of this code, and George Noren for good Java moral support!

Peter Farrett, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Dr. Farrett is the technology leader for the IBM Solution Developer Program's Web site (www.developer.ibm.com) with responsibilities for developing new Web technology applications for Java, chat rooms, multimedia, and assessing future technological directions. He has an MSc and PhD in Computer Science from The City University of New York and Queen's University in Ontario, Canada, respectively.*