Hello World Client and Server Applications

Richard E. Mowe
Computer Networking and Applications Department
St. Cloud State University
mowe@stcloudstate.edu

Abstract

Server and client applications run on the same machine. The server listens on port 8000 for perspective clients. When a client attaches, the server sends the string, “Hello World.” The applications serve as a demonstration of networking concepts and provide skeletons for more useful network programs. Ideas will be offered for further programming assignments.

Introduction

Computer languages provide more or less straightforward ways to handle input and output (I/O) to/from the keyboard, screen, printer, and files. Network I/O, however, is complicated by the overhead of packet management and error checking.

Having to write applications to take packetization and error checking into account is very tedious. Fortunately, the Java programmer can use sockets to communicate among machines and assume that Java, the operating system, or network will handle the nasty details. A socket is a logical device similar to a network drive. Imagine, someone is writing a client application to read data from an existing Web server. One creates a socket to communicate with port 80 on the Web server at a specified address on the network. The programmer then creates input and output streams. Then, data is received from the remote server by reading from the client's input stream. Instructions or uploaded data are sent to the server via the client's output stream. When the client exits, it closes the I/O streams and the socket.

The situation is slightly more complex at the server end. The programmer creates a server socket to *listen* at a designated port on the local machine. Multiple connections are allowed to the server. For each requested connection, a connection socket is created. This is the same type of socket the client uses to connect to the server. I/O streams are opened and data is sent by the output stream and received by the input stream. After the server services the client, it closes the I/O streams associated with that connection and the socket itself (but not the server socket). The server socket stays open to handle connection requests until the server application is shut down.

This paper examines a server application and a client application. The server listens on port 8000 for multiple perspective clients. When a client attaches, the server sends the string, "Hello World." To simplify the situation, both the server and client applications run on the same machine.

The applications serve as a demonstration of networking concepts and provide skeletons for more useful network programs. Ideas will be offered for further programming assignments.

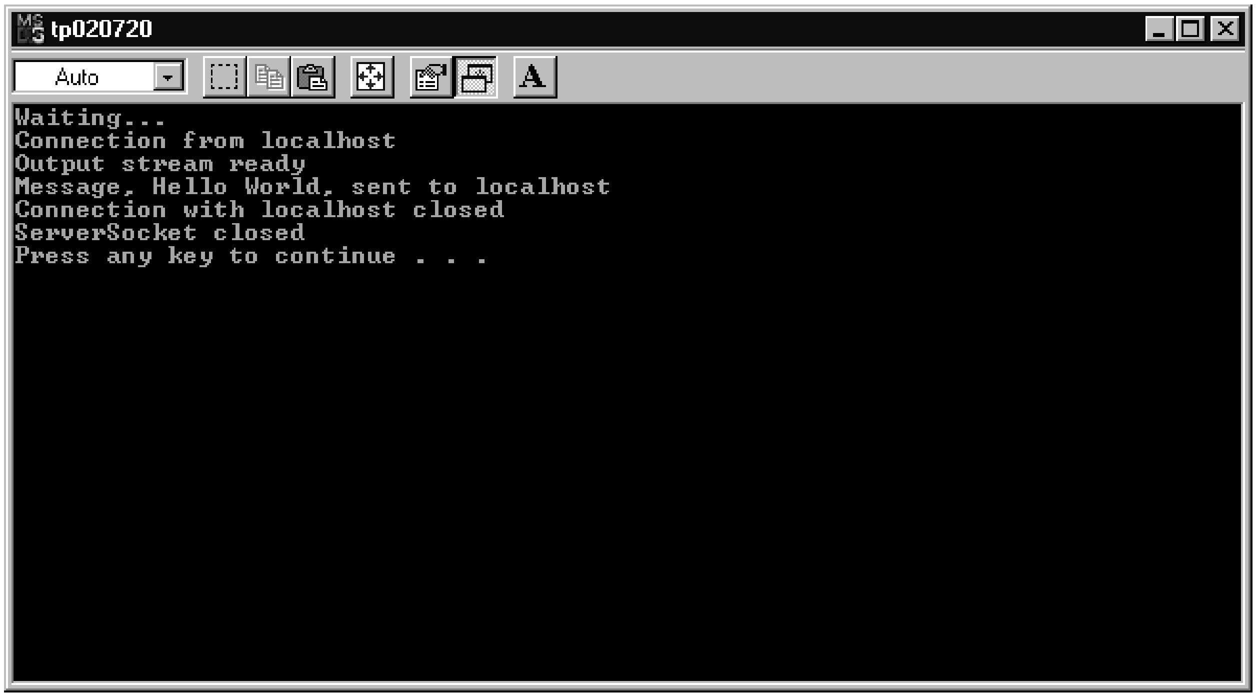
The Java security manager places security restrictions on file and network access for applets. Since the Hello World server and client run as applications, security is not an issue.

Server Application

The server application follows. The server output appears in Figure 1.

```
// HelloServer.java
// Server sends the string "Hello World"
// and closes the connection

import java.io.*;
```

```java
import java.net.*;

public class HelloServer {

   public static void main( String args[] ) {

      ObjectOutputStream outStream;
      ServerSocket sSocket;
      Socket cSocket;
      String clientName;
      String serverMessage = "Hello World";

      try {
         // Create ServerSocket at port 8000 on local machine
         sSocket = new ServerSocket( 8000 );

         // Wait (block) for connection
         System.out.println( "Waiting..." );

         // Accept connection from client
         cSocket = sSocket.accept();
         clientName =
            cSocket.getInetAddress().getHostName();
         System.out.println( "Connection from + clientName "

         // Get output stream
         outStream =
            new ObjectOutputStream( cSocket.getOutputStream() );
         System.out.println( "Output stream ready" );

         // Send Hello World
         outStream.writeObject( serverMessage );
         outStream.flush();
         System.out.println( "Message, + serverMessage +"
            ", sent to " + clientName );

         // Close connection
         outStream.close();
         cSocket.close();
         System.out.println( "Connection with "
            + clientName + " closed" );

         // Close ServerSocket
         sSocket.close();
         System.out.println( "ServerSocket closed" );
         }
      catch ( IOException io ) {
         System.out.println( "I/O error" );
     }
   }
}
```



```
Waiting...
Connection from localhost
Output stream ready
Message, Hello World, sent to localhost
Connection with localhost closed
ServerSocket closed
Press any key to continue . . .
```

Figure 1: Server Output

The program starts by anticipating potential I/O errors in initiating, terminating, writing to, and reading from network connections.

```
      try {
         // Rest of program
         }
      catch ( IOException io ) {
         System.out.println( "I/O error" );
     }
```

Next, the application creates a ServerSocket, which binds to a port and listens (blocks) for connections from clients. Only one server can bind to a given port. Trying to bind to a used port causes an I/O error.

The message, “Waiting...,” shows that the ServerSocket has been created and it is waiting for a connection. Nothing further will happen until a client application attempts a connection to the server.

```
         // Create ServerSocket at port 8000 on local machine
         sSocket = new ServerSocket( 8000 );

         // Wait (block) for connection
         System.out.println( "Waiting..." );
```

When a client connects to the server, a Socket is created by the accept() method of the ServerSocket. It is possible for numerous clients to connect to a server, just as many browsers access a popular web server. A Socket is created on the server for each client that connects. Also, a socket is formed on the client side of the connection. By comparison, only one ServerSocket is created.

The statement clientName = cSocket.getInetAddress().getHostName() is potentially confusing. Let’s take it apart. The statement cSocket.getInetAddress() yields the InetAddress object, which holds the internet address and host name of the client connecting to the server. The getHostName() method returns the name of the calling host and represents it as a string.

```
         // Accept connection from client
         cSocket = sSocket.accept();
         clientName =
            cSocket.getInetAddress().getHostName();
         System.out.println( "Connection from + clientName "
```

Now that the client and server have sockets in place, those sockets must be connected. Input and output streams allow data to flow between client and server. Since the server sends, but does not recieve data, only an output stream is opened. The method getOutputStream() returns a simple output stream object. An output stream by itself is not very useful. It writes bytes and byte arrays, but not objects such as strings and primitive data types such as integers. To gain this functionality, an ObjectOutputStream is constructed from the OutputStream.

```
         // Get output stream
         outStream =
            new ObjectOutputStream( cSocket.getOutputStream() );
         System.out.println( "Output stream ready" );
```

Next, the server sends the message, *Hello World* to the client. The string is written as an object to the ObjectOutputStream. The method flush() ensures that the message is actually sent and not buffered. The client receives the message as an ObjectInputStream.

```
      String serverMessage = "Hello World";
         // Send Hello World
         outStream.writeObject( serverMessage );
         outStream.flush();
         System.out.println( "Message, + serverMessage +"
            ", sent to " + clientName );
```

Finally, the connections are closed. I/O streams are closed and then the Socket. Normally, the ServerSocket stays open to process further connections. In this example, the ServerSocket is closed after one client connection has been processed.

```
         // Close connection
         outStream.close();
         cSocket.close();
         System.out.println( "Connection with "
            + clientName + " closed" );

         // Close ServerSocket
         sSocket.close();
         System.out.println( "ServerSocket closed" );
```

The server application can be tested by telnetting to the server using one of the following commands:

```
telnet localhost 8000
```

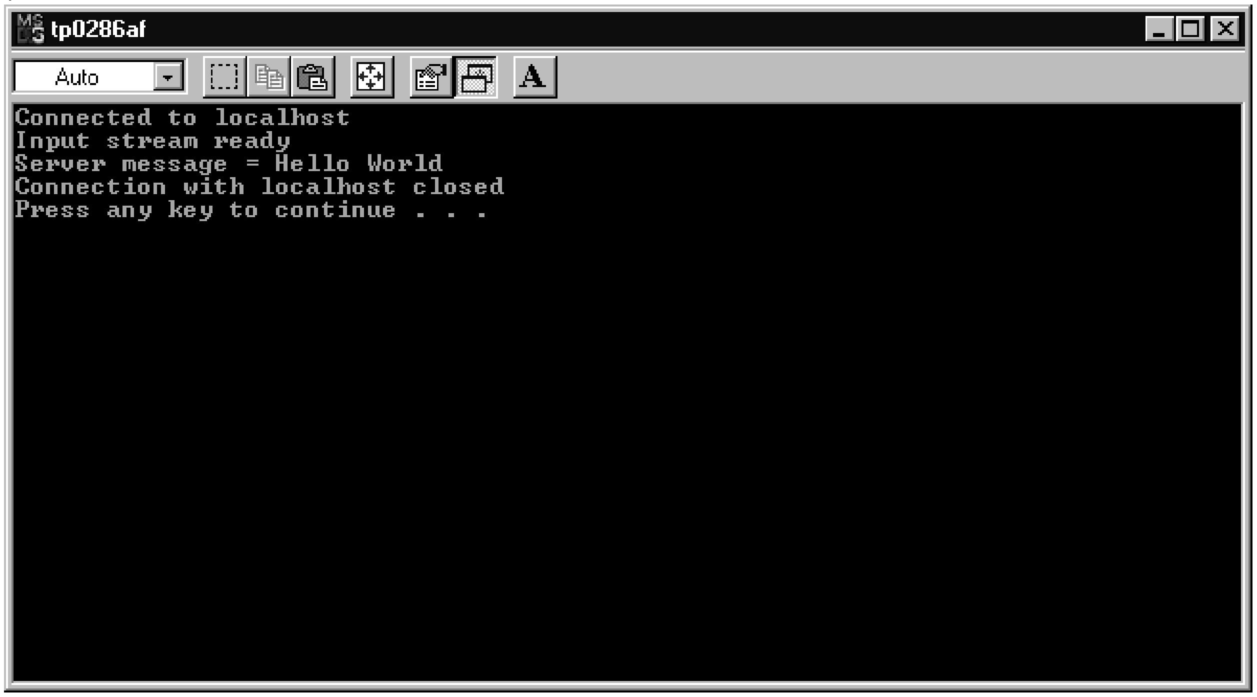
```
telnet 127.0.0.1 8000
```

Either of the commands direct telnet to port 8000 on the local machine. The next section presents a java client to interact with the server.

Client Application

After the server application is started and the message, “Waiting...” appears, then the client application is run. The code follows and output is shown in Figure 2.

```
// HelloClient.java
// Client retrieves a message string from server
// and closes the connection

import java.io.*;
import java.net.*;

public class HelloClient {

   public static void main( String args[] ) {

      ObjectInputStream inStream;
      InetAddress serverAddress;
      Socket cSocket;
      String serverName = "localhost";
      String serverMessage;

      try {
         // Construct InetAddress ojbect
         serverAddress = InetAddress.getByName( serverName );

         // Connect to server by creating socket on server at
         // port 8000
         cSocket = new Socket( serverAddress, 8000 );
         System.out.println( "Connected to " + serverName );

         // Get input stream
         inStream =
            new ObjectInputStream( cSocket.getInputStream() );
         System.out.println( "Input stream ready" );

         // Retrieve server message
         try {
            serverMessage = ( String ) inStream.readObject();
            System.out.println( "Server message = " +
               serverMessage );
         }
         catch ( ClassNotFoundException cnf ) {
            System.out.println( "unrecognized object type" );
         }

         // Close connection
         inStream.close();
         cSocket.close();
         System.out.println( "Connection with " + serverName +
            " closed" );

      }
      catch ( IOException io ) {
         System.out.println( "I/O error" );
      }
   }
}
```



```
Connected to localhost
Input stream ready
Server message = Hello World
Connection with localhost closed
Press any key to continue . . .
```

Figure 2: Client Output

As in the server application, the client application appears within a try block to handle I/O errors. The code will not be shown again.

The client attempts a connection to the server by creating a Socket. The Socket constructor passes the address of the server and the port where the server is listening. The internet address is an InetAddress object, which is constructed using the static InetAddress.getByName() method and passing the host name, “localhost.”

```
      InetAddress serverAddress;
      String serverName = "localhost";

         // Construct InetAddress ojbect
         serverAddress = InetAddress.getByName( serverName );

         // Connect to server by creating socket on server at
         // port 8000
         cSocket = new Socket( serverAddress, 8000 );
         System.out.println( "Connected to " + serverName );
```

Next, an input channel is opened with the server.

```
         // Get input stream
         inStream =
            new ObjectInputStream( cSocket.getInputStream() );
         System.out.println( "Input stream ready" );
```

Then , the readObject() method is used to input a message from the server. The message is received as an Object object and converted to a String Object by the (String) cast. The block is enclosed in a try block to anticipate the potential error of an unknown object type.

```
         // Retrieve server message
         try {
            serverMessage = ( String ) inStream.readObject();
            System.out.println( "Server message = " +
               serverMessage );
         }
         catch ( ClassNotFoundException cnf ) {
            System.out.println( "unrecognized object type" );
         }
```

Finally, the client closes its input stream and connection with the server.

```
         // Close connection
         inStream.close();
         cSocket.close();
         System.out.println( "Connection with " + serverName +
            " closed" );
```

Suggestions

The Hello World server and client can be used as is to demonstrate networking concepts, or can form the basis for programming assignments. As is, they could be used in a networking class to demonstrate the fact that a TCP/IP address includes an IP address and a TCP port number. In a security class, they could show the results of blocking ports in a packet filter.

The client and server applications could be introduced in a network programming class. Students could then modify the programs in exercises such as the following:

1. Write a server to send a list of jokes to the client. Write a client.

2. Write a server to send the date and time. Write a client to receive the date and time and format the output.

Note

This paper is modified from a chapter in the unpublished manuscript, *Auxiliary Java*, which I wrote as a sabbatical project. Its purpose is to provide a Java guide for a course where Java is not the main topic. It is a work in progress. If you would like to see a copy to review or use in a course, please contact me at mowe@stcloudstate.edu.