Simple Clients and Servers in Java

Objectives:

1. Design and implement a simple server in Java.

2. Design and implement a simple client in Java.

3. Add Java threads to a server.

Preparation: Before doing this exercise read the following pages in *Java: How to Program* by Deitel and Deitel, sixth edition: pages 1106-1108 for introduction on networking in Java; pages 1108-1132 on clients and servers, pages 1052-1062 on Java threads.

Exercices:

In this exercise you will construct a Java application for a simple server and a Java application for a simple client which communicates with the server. **NO graphics in the exercise!** Graphics were left out on purpose!

1. The Client Application: Write a Java application which implements a simple client. The client should connect to your server which will be running on another host, e. g., **castor**. Use any **port** above 1024. Note that only one server on a host may use a specific port number. Therefore, if you receive a message of "port in use", just select another port from 1025 to 65535.

In the client, hard code which host will be running your server.

The client should repeatedly read a line a text from **System.in** and send the line to the server. Then the client receives a new line of text from the server and prints it to **System.out**. If the user types exactly "quit", the client sends it to the server to close its connection and the client should close its connection and exit.

Using the example of a client on pages 1117-1132 as a model, write the Java code and compile it. If the server is not running, when you run the client, you should have it print out a message on **System.err** something like "Server is down. Make sure server is running first!".

2. The Server Application: Write a Java application which implements a simple server. Once the socket has been opened, the server waits for the client. After a connection with the client, the server repeatedly receives a line of text from the client, changes the text in some way, and sends it back to the client until the client sends exactly "quit". When the server receives "quit", it sends a usual message to the client, closes the client's connection and waits for another client.

The server should print messages to **System.out** to show what it is doing. For example: "Waiting for a client ..." "Connected with client on lemon".

Using the example of a server on pages 1117-32 as a model, write the Java code and compile it. Run the server first on the proper host, e. g., **castor**, then run your client on a different host.

Note: Your server application should only be able to handle one client at a time. After one client disconnects, the server should be be able to handle a second client and so on. Use **Control-c** to kill your server when needed.

3. Add Java Threads to Server: You are to redo the server application in the above to allow several clients to connect to the server at one time. There is no need for you to modify your client application.

In your new server, you will need to create a new Java thread for each client. Full details of Java threads are covered in Chapter 23 of Java text. However, you don't need to understand all the ins and outs of Java threads to do this exercise.

Use the following code as a model. Study it carefully.

```
// a Java server application with a new thread to serve each new client
// By Dan Hyde, Sept 24, 2001
   import java.net.*;
   import java.io.*;
   public class TCPServer {

      public static void main( String args[] ){

        try {

            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket( serverPort );

            while( true ) {
                System.out.println("Waiting for a client ....");
                Socket clientSocket = listenSocket.accept();
                System.out.println( "Connection received from " +
                                    clientSocket.getInetAddress().getHostName() );
                // start a new thread
                ClientThread c = new ClientThread( clientSocket );
            }
        }
        catch ( IOException e ){
            System.err.println("Error in connection " + e.getMessage() );
        }
      }
   }

   class ClientThread extends Thread {

      ObjectOutputStream out = null;
      ObjectInputStream in = null;
      Socket clientSocket;

      public ClientThread( Socket aClientSocket ){
        try{
            clientSocket = aClientSocket;
            out = new ObjectOutputStream(clientSocket.getOutputStream() );
            in  = new ObjectInputStream( clientSocket.getInputStream() );

            this.start();  // call run()
        }
        catch ( IOException e ){
            System.err.println("Error 1 in connection " + e );
            System.exit(0);
```

```
      }
    }

    public void run() {
        // Assumes client sends a string, receives a string response
        // then quits.
      String lineIn = "";

      try {
          // must cast from Object to String
          lineIn = (String) in.readObject();
          System.out.println( "Received from " +
                              clientSocket.getInetAddress().getHostName() +
                               ": " + lineIn );

          // important to create a new object in this case a String
          out.writeObject(new String( "WOW! " + lineIn ) );
      }
      catch ( ClassNotFoundException e ){
          System.err.println( "Class not found " + e );
          System.exit(0);
      }
      catch ( IOException e2 ){
          System.err.println( "Error in I/O " + e2 );
          System.exit(0);
      }

      // close client
      try {
          in.close();
          out.close();
          clientSocket.close();
          System.out.println( "Connection closed from " +
                              clientSocket.getInetAddress().getHostName() );
      }
      catch ( IOException e3 ){
          System.err.println( "Error 2 in connection " + e3 );
          System.exit(0);
      }
    }

}
```

One way to create threads is to have a class that `extends Thread`. Any class that extends the class `Thread` must have a `public void run()` method. You start the thread by calling the `start()` method of the super class `Thread` which calls your `run()` method.

The idea of the exercsie is to have separate threads that continue to process existing clients while the server waits for requests from new clients. When the server accepts a new client, the server creates a new thread to handle it and goes back to waiting for a new client.

Using Java threads for your server is relatively easy because the different threads do not share information nor need to coordinate between themselves. Having a server spawn a new thread (or process) for each new client is very common in client/server applications.