CSCI 355 LAB #5 Spring 2007

Adding Java Threads to a Server; Java Serialization

Objectives:

1. Add Java threads to a server.
2. Explore Java serialization.

Preparation: Before lab read the pages on **Serializable** interface in *Java: How to Program* by Deitel and Deitel.

Study Figure 4.6 in Distributed Systems Text.

Bring the Java and Distributed Systems texts to lab.

Laboratory Assignment:

Exercise 1: Add Java Threads to Server

Introduction

A very common technique of servers is to create a new thread to handle each new client. This allows a server to service many clients simultaneously. If a client quits, the server's thread associated with the client is terminated. This is a clean way to deal with client termination.

In this lab, since the threads associated with each client are completely independent of each other, this makes the threads relatively easy to program.

Details

You are to redo the server in Lab 4 to allow several clients to connect to the server at one time. There is no need for you to modify your client application of Lab 4.

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 lab. We will do more with Java threads in later labs.

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("From Server: " + 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);
        }
      }
   }
```

This code is close to the code in Figure 4.6 on page 143 in our Distributed Systems text.

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 lab 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.

Change the server to send back the same string but in all lower case.

Exercise 2: Java Serialization

Introduction

A stream is a sequence of bytes. In order to send an arbitrary object across **ObjectOutputStream()**, Java automatically converts where needed, packs and labels the different pieces of the object into a sequence of bytes. Java calls this process *serialization*. At the other end, the receiver must unpack and recreate an object that has same state and behavior as the original object. This is called *deserialization*. Other programming languages call this *marshalling* and *de-marshalling* arguments.

Details

Start with fresh copies of your client and server files from Lab 4 (NOT copies of files from Exercise 1). The idea of this exercise is to practice passing a Java object on **ObjectOutputStream()** and **ObjectInputStream()**. In this case you will pass a user-define **Exam** object.

Create a new class **Exam** in a *separate file* called **Exam.java**. Probably your **Exam** class from Lab 2 will be close to what you need. This new class should have two **private** instance objects **name** and **score**. Write **Get** and **Set** methods to allow others to access the instance objects. The class must implement **Serializable** interface. This tells the compiler to generate code such that an **Exam** object can be serialized for an **ObjectOutputStream()** stream. See Java text for how.

Alter the client to allow the user to enter a name and a score and create an **Exam** object. This **Exam** object is sent to the server. The server should double the score and return a new altered **Exam** object. When the client receives an **Exam** object, it should display the name and the new score to the screen. The client loops to allow many **Exam** objects to be sent, modified and returned.

Note: Within an **ObjectInputStream()** stream, the *first* reference to any object results in the object being *serialized* and the assignment of a *handle* (special reference) for that object. Subsequent references to that object are encoded as only the handle. This means that it is important for you to send new objects in **out.writeObject()** and not an existing object with changed state (values). We recommend something like:

```
        out.writeObject(new Exam(name1, score1));
```

which assumes you have a two argument constructor.

Note: In last week's lab, objects of type **String** behave such that we did not need to concern ourselves with this issue. In Java, **String**s are immutable, i.e., can't change their values.

Hand In:

For Exercise 1, show your lab instructor that it works with three different clients on different hosts at the same time. Hand in a printout of the new server code and a run showing the server output and one client's output.

For Exercise 2, hand in the Java code for the three files and a sample output of the client and server that shows at least 5 **Exam** objects were sent.