**ISTE-121 Day20 – Data Transfer and Multi-Threaded TCP Server**

**Data Transfer**

Last class, we sent String objects using a DataOutputStream and received them using a DataInputStream. Today, we’ll look at how to transfer a file one byte at a time using these streams.

We’ll be modifying TCPClient.java and TCPServer.java, in today’s downloads, so that the client sends a file called test.txt (also in today’s downloads) to the server byte by byte. To do this, we’ll be using a writeByte to send data and readByte to receive data.

Before you get too far, rename TCPClient to TCPClientData and TCPServer to TCPServerData.

**Modifying the Client and Server**

Modify the client GUI so that the text label says “File:” instead of “Sentence:” and the text field should display test.txt all the time and cannot be modified by the user.

After connecting to the server, when the user clicks the Send button, the doSend method should open the test.txt file by first creating a File object called myFile, then creating a DataInputStream to read the file.

DataInputStream fin = new DataInputStream(new FileInputStream(myFile));

Upon opening the input stream, the client will determine the length of the file and send the length to the server with the writeLong method. Note that the file length is reported as a long. The server will then respond with the file length it received from the client. Once the client receives the message back from the server, the client will send the file one byte at a time to the server using the writeByte method. To get a byte from the file, the client must use readByte on fin. As the server receives bytes, it should output to the console the number of the byte it has received.

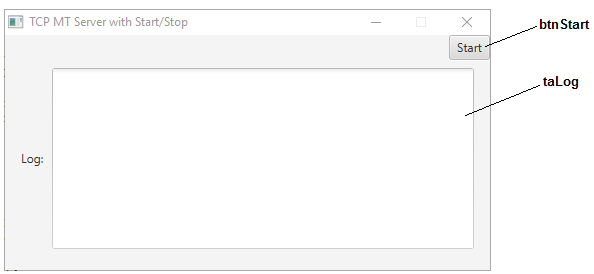
When the server has received all of the bytes of the file, it should reply with a String (writeUTF) that says “DONE”, so that the client knows the transfer is complete. At this point, the client and server should flush their output streams using out.flush() and the client will close the file it was transferring. The server will close the connection and its streams. What should the client do now that the server has closed the connection?

To properly implement and test this code, you will need to make gradual changes to the client and server. Use lots of log messages to show your progress!

**Multi-Threaded TCP Server**

We’ll use TCPServer.java, in today’s downloads, as a starting point for creating a multi-threaded version of the TCP server. For testing, you’ll connect to the server using TCPClient.java.

**Improvements**Rename TCPServer.java to TCPServerMT.java. Allow the server to be started and stopped. Add a Start/Stop button to the server GUI (one button that can be either Start or Stop, like Connect/Disconnect in the client). This should be added in a FlowPane that is added to the root VBox before the FlowPane containing the Log Label and the taLog TextArea. You may also need to increase the height of the scene. On Start, start the loop with accept. We’ll talk about Stop below:



Create a new (inner-named) thread class, called ServerThread. This class will (inside run()) create the ServerSocket and accept new connections (the accept loop). However, the accept loop will be shortened somewhat (below).   
 class ServerThread extends Thread {  
 public void run() {  
 // Server stuff ... wait for a connection and process it  
 try {  
 sSocket = new ServerSocket(SERVER\_PORT);  
 }  
 catch(IOException ioe) {  
 // "log" discussed below  
 log("IO Exception (1): "+ ioe + "\n");  
 return;  
 }  
   
 while(true) {  
 // Socket for the client  
 Socket cSocket = null;  
   
 try {  
 // Wait for a connection and set up IO  
 cSocket = sSocket.accept();  
 }  
 catch(IOException ioe) {  
 // Happens when sSocket is closed in the stop (below)  
 // and the accept (above) is blocked. This is OK.  
 // Log the error then return  
 return;  
 }   
   
 // Create a thread for the client, passing cSocket to the  
 // thread’s constructor and start the thread...  
 ClientThread ct = new ClientThread(cSocket);  
 ct.start();  
 } // of while loop  
 } // of run

**Note:** The call to the **log** method above, will be to the following private method inside the main, Application, class.

// utility method "log" to log a message in a thread safe manner  
 private void log(String message) {  
 Platform.runLater(new Runnable() {  
 public void run() {  
 taLog.appendText(message);  
 }  
 } );  
 } // of log

Use of this method will avoid having to retype the Platform.runLater() … code everytime we wish to write to the Log TextArea from within another thread.

When the Start button is clicked, create and start this thread (ServerThread) running. When the Stop button is clicked, call the stopServer method of the ServerThread to stop the thread. One way to stop the thread is to close the ServerSocket in the stopServer() method. This will cause an IO Exception in the run() method if we are blocked at accept(), which can be logged and then cause run() to return.

public void stopServer() {  
 try {  
 sSocket.close(); // This terminates any blocked accepts  
 }  
 catch(Exception e) {  
 log("Exception: " + e + "\n");  
 }  
 }

Create another new (inner-named) thread class called ClientThread. This class will represent a thread that will be started for each client.

class ClientThread extends Thread {

} // End of inner class

Inside the accept loop of the ServerThread, instantiate and start a ClientThread. This thread will need the Socket that is returned by accept … it should be passed to the ClientThread constructor and saved in an attribute of the ClientThread class.

// Create a thread for the client  
 // Done after the red comments above  
 ClientThread ct = new ClientThread(cSocket);  
 ct.start();

In ClientThread’s run() method is where you should open the streams for IO and do all the work for the client. When the client is done (hasNextLine() returns false), close the Socket and the streams and terminate the thread (return from the run() method).

class ClientThread extends Thread {  
 // Since attributes are per-object items, each ClientThread has its OWN  
 // socket, unique to that client  
 private Socket cSocket;  
 private String clientId = "";  
  
 // Constructor for ClientThread  
 public ClientThread(Socket \_cSocket) {  
 cSocket = \_cSocket;  
 clientId = cSocket.getInetAddress().getHostAddress() + ":" + cSocket.getPort();  
 }  
   
 // main program for a ClientThread  
 public void run() {  
 DataInputStream in = null;  
 DataOutputStream out = null;  
   
 log(clientId + " Client connected!\n");  
   
 try {  
 // Open streams   
 in = new DataInputStream(cSocket.getInputStream());  
 out = new DataOutputStream(cSocket.getOutputStream());   
 }  
 catch(IOException ioe) {  
 log(clientId + " IO Exception (ClientThread): "+ ioe + "\n");  
 return;  
 }  
   
 try {  
 String line = in.readUTF();  
 taLog.appendText("Received: " + line + "\n");  
 line = line.toUpperCase();  
 taLog.appendText("Replying: " + line + "\n");   
 out.writeUTF(line);  
 }  
 catch(Exception e) {  
 taLog.appendText("Error during transmission: " + e + "\n");  
 }  
   
 // on EOF, client has disconnected   
 try {  
 // Close the Socket and the streams (ADD THE CODE)  
 }  
 catch(IOException ioe) {  
 log("IO Exception (3): "+ ioe + "\n");  
 return;  
 }  
   
 log("Client disconnected!\n");  
 }   
 } // End of inner class

***With this new design, we create a new thread to handle each client. This means that the accept loop continues on to accept a new connection (if one is waiting) without waiting for the client to finish. We can now serve many clients at the same time … each in its own thread.***

***Also, have each ClientThread report its own IP address and port number:*** 129.21.4.23:32005  
and announce that a client has connected. This should be near the start of the run() method.

Prepend all strings to the log with the IP:PORT string for that client, so we know what client is causing the logging. You can find out a client’s IP and PORT from the Socket object that is created when the client is accepted. Look up in the API the getInetAddress() method of the Socket class. Also, look up the getPort() method. These are a start, but for the IP address, you need to drill down a little further. Look at the getInetAddress() method of Socket, then, in the InetAddress class, check out the getHostAddress() method to get the IP address.