Exercise Session 6: Creating a Distributed Application

COM-208: Computer Networks

The goal of this lab is to get hands on experience with designing and building a distributed application.

To do the lab, you first need to know how to compile and run a Java program:

```
javac filename.java
java filename
```

(Of course, you are free to use any graphical interface you are used to.) You also need basic knowledge of Java programming. The following could help you freshen up your Java skills:

- Java Tutorials, in particular the I/O Tutorial and DataInputStream tutorials.
- A Java Socket Programming Tutorial.

You are given a simple application (that runs on a single computer), which you will convert it to a client-server application that uses TCP as its transport-layer protocol. We will first describe the original application, then the final application that you need to design and build, and then a series of milestones (steps) that you could follow to reach your final goal. At the end of the lab description, you will find hints.

The original application

The application receives input from the user through the command prompt.

If the user enters an empty filename (i.e., simply presses the "Enter" key) or an invalid filename (i.e., the file does not exist), the application exits.

While if the user inputs a valid filename (which is located in the same folder as the application), the application prints certain statistics about the contents of the file. Then, the application waits for the user to input a new filename.

In particular, the application prints the following stats about the contents of the file:

- The length of the file, in bytes.
- The number of different unique words in the file.
- A list of words and how many times each word appears in the file.

The following snippet, demonstrates an example run of the application. The first time, the user inputs "ppap.txt", and the application prints word-occurrence stats about the file. In this case, the "ppap.txt" file was in the same folder as the Java application. The second time, the user inputs nothing. Thus, the application terminates.

```
Enter a file name: ppap.txt
The file has length: 219 bytes
There are 10 unique words in the document

a: 4
apple: 5
have: 4
i: 4
intro: 1
p: 3
pen: 10
pineapple: 5
uh: 3
verse: 3
Enter a file name:
(The program terminates)
```

The source code for this application is in `WordCounter.java`.

Client-server application description

You will create two files, `TCPClient.java` and `TCPServer.java`. As the names indicate, one will implement a client and the other will implement a server. Here are their responsibilities:

Client

- The client is responsible for reading the user input.
- Every time the user inputs a new filename, the client reads the file content and sends it to the server.
- Then, the client waits until the server returns its response.
- When the response from the server arrives, the client prints the results to the user.
- Then, the client prompts the user for a new filename.
- Whenever the user enters an empty or invalid input, the client notifies the server that the session is complete and terminates its TCP connection with the server.

Here is a piece of what your client might look like:

```java
// package ch.epfl.compnet;

import java.io.*;
import java.nio.file.*;
import java.net.*;
import java.util.*;

public class TCPClient {
    private static FileInputStream getFileReader(String filename) {
        FileInputStream fis = null;
        boolean fileExists = true;
        try {
            fis = new FileInputStream(filename);
        } catch (FileNotFoundException e) {
            fileExists = false;
        }

        return fis;
    }

    private static int getFileLength(String filename) {
        File file = new File(filename);
        int length = (int) file.length();

        return length;
    }
```

```java
    private static void printHashMap(Map<String, Integer> occurrences) {
        for (String name: occurrences.keySet()) {
            String key = name.toString();
            String value = occurrences.get(name).toString();

            System.out.println(key + " " + value);
        }
    }

    public static void main(String argv[]) {
        //@TODO
    }
}
```

Server

- The server waits for TCP connections from clients at port 6789.
- When a client connects, the server waits to receive the file contents from the client.
- Once the server finishes receiving the file, it processes its content and transmits the results (the same statistics as the original application) back to the client.
- Then, without closing the TCP connection, the server waits for new input from the client.
- If, at any moment, the client notifies the server that their session is over, the server terminates the TCP connection.
- The server itself never terminates, since it may have to serve additional clients.

Here is a piece of what your server might look like:

```java
// package ch.epfl.compnet;

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


public class TCPServer {
    private static Map<String, Integer> getOccurrences(String message) {
        Map<String, Integer> occurrences = new TreeMap<String, Integer>();

        String delimiterRegexp = "[^a-zA-Z]+";

        Scanner fileScan = new Scanner(message).useDelimiter(delimiterRegexp);

        while(fileScan.hasNext()){
            String word = fileScan.next();
            word = word.toLowerCase();
```

```java
            Integer oldCount = occurrences.get(word);
            if ( oldCount == null ) {
                oldCount = 0;
            }
            occurrences.put(word, oldCount + 1);
        }

        fileScan.close();
        return occurrences;
    }

    public static void main(String args[]) {
        //@TODO
    }
}
```

Hints before you start

1. Creating socket streams.

In Java, to write and read data to/from a socket, you need to first obtain references to the input and output streams of the `Socket` object of the connection. You will use the input stream to read data from the other host, and the output stream to send data to other host.

You can read more about the available stream options at the Java I/O Tutorial. Two pairs of classes you will find of particular use for this project are the following:

- `BufferedReader`/`BufferedWriter` (for text data)
- `DataInputStream`/`DataOutputStream` (for binary data)

2. Blocking operations

If your client/server tries to read too much from a socket, the host will permanently block until the other end transmits enough information.

To avoid unnecessary trouble, it is *extremely* important that you plan out you protocol thoroughly, in advance. Try to create a sequence diagram about exactly what each host expects from the other at every part of the connection.

Most importantly, you need to make sure that the two ends of the communication will never get stuck waiting for each other at the same point in time. Needless to say, this would lead in a "deadlock".

Note that for your application you need to account for 3 communicating ends:

- The server
- The client
- The user that inputs filenames on the client console

3. Figuring out the right format for your messages.

This is very strongly related to what has already been stated in Hint #2.

When sending messages using TCP, the boundaries between consecutively sent messages get lost. So, You need to make sure that your messages do not get misinterpreted!

That is to say, if you use a TCP socket to transmit "Hello" and "Goodbye" as two separate messages, the other may interpret this as one single message: "HelloGoodbye". This is because all data transmitted using TCP gets "serialized" into a single byte-stream.

Thus, you need to construct your messages in a way such that you can deserialize the byte-stream back to the original messages. For instance, if you know that the character "|" can never be part your message, you can transmit "Hello|Goodbye" to make the remote end understand that those are two different messages. In this case, the role of "|" is that of a delimiter.

If there is no character that can act as a delimiter for your protocol, you need to think about a way to construct headers for your messages. Then, these headers can be used by the other end to deserialize your messages.

Implementation milestones

Create your application piece by piece, not all at once. This will make it easier to test
your code as you go and avoid nasty surprises when your code becomes more complex.

1. The client transmits one file to the server.

Assume that the client has only one file to transmit to the server. When the client finishes
transmitting the file, the client terminates. When the server finishes receiving the file, it
prints the statistics and also terminates. Below is a template of the methods you need to
add or modify for each file.

```java
//TCPClient.java

private static boolean sendFile(DataOutputStream os, String filename) throws
    ↪ IOException {
  boolean retval;

  if (filename.isEmpty() || !Files.isReadable(Paths.get(filename))) {
      retval = false;
      ...
  } else {
      retval = true;

      // Send the file itself
      FileInputStream fis = getFileReader(filename);
      ...
  }
  return retval;
}

public static void main(String argv[]) {
  // Connect to the local server at 6789
  Socket clientSocket = ...;

  BufferedReader inFromUser = ...;
  DataOutputStream outToServer = ...;

  ...

  // sendfile will notify us whether this is the final file or not
  sendFile(outToServer, filename);
}
```

```java
//TCPServer.java

private static void sendResponse(DataOutputStream outToClient, String message)
    ↪ {

  // Perform word-occurrence stats
  Map<String, Integer> occurrences = ...;
  ...
}

private static void handleConnection(Socket connectionSocket) {
  DataInputStream inFromClient = null;
  DataOutputStream outToClient = null;

  // Open the input-output streams
  ...

  // Read the file contents into message
  byte[] bytearray = ...
  ...

  // Call the response handler
  sendResponse(outToClient, message);
}

public static void main(String args[]) {
  // Create a socket that listens to port 6789
  ServerSocket welcomeSocket = ...;

  // Get a new connection
  System.out.println("Waiting for a new client...");
  Socket connectionSocket = ...;

  // pass the connection socket to the handler
  System.out.println("Handling new client...");
  handleConnection(connectionSocket);
}
```

2. The client transmits multiple files to the server, even if it doesn't always work.

Assume that some character (e.g., "|") never appears in any file. This enables the client to use this character to signal file boundaries to the server.

```java
//TCPClient.java

private static boolean sendFile(DataOutputStream os, String filename) throws
    ↪ IOException {
  boolean retval;
```

```java
    if (filename.isEmpty() || !Files.isReadable(Paths.get(filename))) {
        retval = false;
        ...
    } else {
        retval = true;

        // Send the file itself
        ...

        // Send the special character
        ...
    }
    return retval;
}


//TCPServer.java

private static void handleConnection(Socket connectionSocket) {
  DataInputStream inFromClient = null;
  DataOutputStream outToClient = null;

  // Open the input-output streams
  ...

  // Read the file contents into message
  byte[] bytearray = ...
  ...

  // Process the special character
  ...

  // Call the response handler
  sendResponse(outToClient, message);
}
```

3. The server responds with part of the statistics.

The server now sends the statistics to the client instead of printing them. Assume that the server responds with a fixed amount of information (e.g., information about only 5 words). This means that the client knows how much information to anticipate from the server, hence how much to read.

```java
//TCPClient.java

private static void handleResponse(DataInputStream inFromServer) throws
    ↪ IOException {
```

```java
    int numValues = 5;

    System.out.println("There are " + numValues + " unique words in the document
      ↪ \n");

    for (int i = 0; i < numValues; i++) {
        // Read the length of the word
        int length = ...

        // Allocate a big enough buffer for the word
        byte[] bytearray = new byte[length];

        // Actually read the word and convert it to a string
        ...
        String word = new String(bytearray);

        // Read the number of occurrences
        int times = ...

        System.out.println(word + ": " + times);
    }
}

//TCPServer.java

private static void sendResponse(DataOutputStream outToClient, String message)
    ↪ throws IOException {
  // Perform word-occurrence stats
  Map<String, Integer> occurrences = ...

  int numValues = 5;

  for (String key: occurrences.keySet()) {
      String word = key.toString();
      int times = occurrences.get(key);

      // Send the length of the word first
      ...

      // Then, send the actual word
      ...

      // Finally, send the number of times the word appears
      ...

      // Break when already sent 5 words
      ...
  }
}
```

4. The server responds with the complete information.

Remove your previous assumption that the client magically knows how much information to anticipate from the server. Design a way for the server to tell the client how much to read.

```java
//TCPClient.java

private static void handleResponse(DataInputStream inFromServer) throws
    ↪ IOException {
  int numValues = ...;
  ...
}
```

```java
//TCPServer.java

private static void sendResponse(DataOutputStream outToClient, String message)
    ↪ throws IOException {

  // Perform word-occurrence stats
  Map<String, Integer> occurrences = ...

  // Send the number of words
  int numValues = ...
  ...
}
```

5. Eliminate special characters and close TCP connections.

Let's try to do things without a special character. Design a proper way (think headers) for client to: (1) signal file boundaries to the servers, and (2) signal server to terminate its TCP connection when the user enters an empty or invalid input. Note that both client and server should gracefully close their TCP connections when they terminate.

```java
//TCPClient.java

private static boolean sendFile(DataOutputStream os, String filename) throws
    ↪ IOException {
  boolean retval;

  if (filename.isEmpty()) {
      retval = false;
      os.writeInt(0);
  } else {
      retval = true;

      // Send the file length
```

```java
        int length = ...
        System.out.println("The file has length: " + length + " bytes");
        ...

        // Send the file itself
        ...
    }
    return retval;
}

public static void main(String argv[]) {
    Socket clientSocket = null;

    BufferedReader inFromUser = null;
    DataOutputStream outToServer = null;
    DataInputStream inFromServer = null;


    Boolean repeatFlag;

    try {
        // Connect to the local server at 6789
        clientSocket = ...

        inFromUser = ...
        outToServer = ...
        inFromServer = ...

        System.out.println("Connected to server");
        do {
            System.out.print("Enter a file name: ");
            String filename = inFromUser.readLine();

            // sendfile will notify us whether this is the final file or not
            repeatFlag = sendFile(outToServer, filename);
            if (repeatFlag == true) {
                // If we didn't send a file,
                // we don't need to wait for a response
                handleResponse(inFromServer);
            }

        } while(repeatFlag == true);
    } catch (IOException ioex) {
        System.out.println("Failed to process request : " + ioex.getMessage());
    } finally {
        // Close all input/output/sockets
        ...
    }
}
```

```java
//TCPServer.java

private static void handleConnection(Socket connectionSocket) {
  DataInputStream inFromClient = null;
  DataOutputStream outToClient = null;

  try {
      // Open the input-output streams
      inFromClient = ...
      outToClient = ...

      // This variable controls when the loop should terminate
      boolean repeatFlag = true;

      do {
          // Read the length of the file
          int length = ...
          System.out.println("The file has length: " + length + " bytes");

          if (length == 0) {
              // Terminate the connection
              repeatFlag = false;
          } else {
              // Read the file contents into message
              byte[] bytearray = new byte[length];
              ...
              String message = new String(bytearray);
              System.out.println(message);

              // Call the response handler
              sendResponse(outToClient, message);
          }
      } while (repeatFlag == true);

  } catch (IOException ioex) {
      System.out.println("Failed to handle connection : " + ioex.getMessage());
  } finally {
      // Close all input/output/sockets
      ...
  }
}
```

6. (Optional) The server disconnects clients that take too long to respond.

So far, your server serves only one client at a time. What if:

- a single malicious client connects to your server and never does anything?

- a client only manages to transmit a few bits before it crashes?

The server should be able to free its resources and serve other clients. To do this, you need to make reads non-blocking.

```java
//TCPServer.java

private static void handleConnection(Socket connectionSocket) {
  DataInputStream inFromClient = null;
  DataOutputStream outToClient = null;

  try {
      // Set reads to timeout after 50 seconds (50000 milliseconds)
      ...

      // Open the input-output streams
      inFromClient = ...
      outToClient = ...

      ...
  } catch (IOException ioex) {
      ...
  } finally {
      // Close all input/output/sockets
      ...
  }
}

public static void main(String args[]) {
  ServerSocket welcomeSocket = ...;
  Socket connectionSocket = ...;

  try {
      // Create a socket that listens to port 6789
      welcomeSocket = ...;

      while(true) {
          try {
              // Get a new connection
              System.out.println("Waiting for a new client...");
              Socket connectionSocket = ...;

              // pass the connection socket to the handler
              System.out.println("Handling new client...");
              handleConnection(connectionSocket);
          } catch (IOException ioex) {}
      }
  } catch (IOException ioex) {
      System.out.println("Failed to open welcomeSocket: " + ioex.getMessage());
  } finally {
```

```
        // Close welcomeSocket
        ...
    }
}
```

7. (Optional) The server serves multiple clients at the same time.

So far, your server serves only one client at a time, which means that new clients have to queue up and wait for the currently-active client to disconnect.

Experiment with threads. Implement a thread-based server, where each thread handles another client.

```
//MultiThreadServer.java

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


public class MultiThreadServer {
    public static void main(String args[]) {
        ServerSocket welcomeSocket = null;
        Socket connectionSocket = null;

        try {
            // Create a socket that listens to port 6789
            welcomeSocket = ...

            while(true) {
                try {
                    // Get a new connection
                    connectionSocket = ...

                    // Start a new thread for the accepted connection
                    ...
                } catch (IOException ioex) {}
            }
        } catch (IOException ioex) {
            System.out.println("Failed to open welcomeSocket : " + ioex.
    ↪ getMessage());
        } finally {
            // Close inputs/outputs/sockets
            ...
        }
    }
}
```

```java
class RequestHandler implements Runnable {
    Socket socket;

    public RequestHandler(Socket socket) {
        this.socket = socket;
    }

    ...

    public void run() {
        ...
    }
}
```