Computer Networks

TP5: Socket Programming - TCP

Objectives:

- Acquire first-hand experience about how to develop an application-layer protocol.
- Learn to identify and avoid common pitfalls in network application design.
- Learn socket programming in Java: TCP

Prerequisites:

- Lectures 3, 4 and 5
- How to compile and run a Java program (your IDE might have a GUI to run):
```
javac filename.java
java filename
```
- Basic knowledge of Java programming. If some mechanisms or concepts are unclear, you may find help and examples in:
  - Java Tutorials, in particular the I/O Tutorial and DataInputStream tutorials.
  - A Java Socket Programming Tutorial

Note: The example codes below are for illustration only. You must take care of compiling and running successful yourself.

Goal

In this TP you will be given a simple Java application, which you will have to convert to a Client-Server application. For this TP, we will be using TCP.

In this document we will:

- First give a brief description of the original Java application (that runs on one machine)
- Then, we describe how the application should look in its final form (Client-Server application).
- Finally, we will provide a series of steps (milestones) which you can follow to reach the final goal.

Original Application

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

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

Otherwise, if the user enters an empty filename (i.e., simply presses the "Enter" key), the application exits.

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 that exist in the document
- a list of words and how many times these words appeared in the document

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)
```

Here is the source code for the application, which you will have to modify:

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

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

public class WordCounter {

    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;
    }

    private static Map<String, Integer> getOccurrences(String filename) {
        Map<String, Integer> occurrences = new TreeMap<String, Integer>();

        String delimiter_regexp = "[^a-zA-Z]+";
        FileInputStream fis = getFileReader(filename);


        Scanner fileScan = new Scanner(fis).useDelimiter(delimiter_regexp);

        String word;
        while(fileScan.hasNext()){
            word = fileScan.next();
            word = word.toLowerCase();
            Integer oldCount = occurrences.get(word);
            if ( oldCount == null ) {
                oldCount = 0;
            }
            occurrences.put(word, oldCount + 1);
        }

        fileScan.close();
        return occurrences;
    }

    private static void printMap(Map<String, Integer> occurrences) {
        int num_values = occurrences.size();

        System.out.println("There are " + num_values + " unique words in the document \

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

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

    private static void printFileStats(String filename) {
        int length = getFileLength(filename);
        Map<String, Integer> occurrences = getOccurrences(filename);

        System.out.println("The file has length: " + length + " bytes");
        printMap(occurrences);
    }

    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in)
        boolean repeatFlag;

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

            if (filename.length() > 0) {
                printFileStats(filename);
                repeatFlag = true;
            } else {
                repeatFlag = false;
            }
        } while(repeatFlag == true);
    }
}
```

Client-Server application

For this exercise you will create two files named `TCPServer.java` and `TCPClient.java`.

Your goal is to convert the single-host application that was given to you, and convert it to a Client-Server application. In particular, you need to partition the operations of the client and the server as follows:

Client:

- The client is responsible for reading input from the user (filenames)

- Every time the user inputs a new filename, the client reads the file contents (text) and streams them to the server
- Then, the client waits until the server returns its response.
- When the response from the server arrives (statistics), the client prints the results to the user
- Then, the client prompts the user for a new filename to send to the server
- Whenever the user enters an empty input, the client needs to notify the server that the session is complete. Then, the client terminates its connection to the server..

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

import java.io.*;
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;
    }

    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 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 contents and transmits the results back to the client.
- Then, without closing the connection, the server waits for new input from the client.
- If, at any moment, the client notifies the server that they have no more files to send (empty input), the server terminates its connection to the client.
- The server never terminates, since it may have to serve additional clients

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

import java.util.*;
```

```java
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 delimiter_regexp = "[^a-zA-Z]+";

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

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

            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
    }
}
```

Milestones

Do **not** try to implement everything at once. Instead, try to structure your application piece-by-piece. This will make it easier for you to test your code more thoroughly, and avoid nasty surprises when your code becomes more complex later on.

1. Try to get one file-transfer done right, first.

For this, you assume that the client only has one file it wants to transmit to the server.

When the client finishes transmitting the file, the client will shut down. Once the server finishes receiving the file, it will print the file contents and also shut down.

```java
//TCPClient.java

private static boolean sendFile(DataOutputStream os, String filename) throws IOExceptio
  boolean retval;

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

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

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

2. Try to think of a way to allow the client to transmit multiple files to the server, even if it isn't completely correct.

For this, you can assume that some character is never allowed to show up inside a file (e.g., "|"). Thus, the client can use the "|" character to signal file boundaries.

```
//TCPClient.java

private static boolean sendFile(DataOutputStream os, String filename) throws IOExceptio
  boolean retval;

  if (filename.isEmpty()) {
      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
```

```
      send_response(outToClient, message);
   }
```

3. Try to make the server respond with only a part of the useful information (e.g., information about only 5 words)

If you can anticipate how much information to expect from the server, your client will know exactly how much to read.

```
//TCPClient.java

private static void handleResponse(DataInputStream inFromServer) throws IOException {
  int num_values = 5;

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

  for (int i = 0; i < num_values; 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 send_response(DataOutputStream outToClient, String message) throw
    // Perform word-occurrence stats
    Map<String, Integer> occurrences = ...

    int num_values = 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. Extend your server-client communication so that the server can transmit information about all the words.

The server program needs to transmit information about all words encountered in the file; not just 5 words.

To avoid a "deadlock" situation, you need to think of a way to make the client know about how much it should read.

```java
//TCPClient.java

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

```java
//TCPServer.java

  private static void send_response(DataOutputStream outToClient, String message) throw

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

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

5. Find a way to terminate the connection.

Once the client no longer requires anything from the server (empty input), both the server and the client should be able to terminate gracefully (and not just the client).

```java
//TCPClient.java

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
        ...
    }
}
```

```
//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
                send_response(outToClient, message);
            }
        } while (repeatFlag == true);

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

6. Make client-server communication less reliant on special characters.

Make it so that your protocol no longer relies on special characters to signal message boundaries. You need to devise proper headers for every file transmitted.

```
//TCPClient.java

private static boolean sendFile(DataOutputStream os, String filename) throws IOExceptio
  boolean retval;

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

      // Send the file length
      int length = ...
      System.out.println("The file has length: " + length + " bytes");
      ...

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

```
//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 {
            System.out.println("Waiting to receive a file...");

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

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

                // Call the response handler
                send_response(outToClient, message);
            }
        } while (...);

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

(Optional) 7. The server should disconnect clients that take too long to respond.

Currently your server only allows one client to be active at a time. However:

- What if a single malicious client connects to your server and never does anything?
- What if a client only manages to transmit a few bits before it crashes?

Your server should be able to free its resources to be used by other clients. To do this, you need to find a way to make reads non-blocking.

```
//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
      ...
  }
}
```

(Optional/Advanced) 8. Make your server able to serve multiple clients at the same time.

If you have implemented everything correctly so far, your server will only be able up to one client at the same time. New clients will have to queue up and wait for the currently-active client to disconnect.

If you feel comfortable with threads, you can try implementing a thread-based server. Each thread will handle a separate client connection.

```
//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
            ...
        }
    }
}

class RequestHandler implements Runnable {
    Socket socket;

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

    ...

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

Hints

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 of these messages get lost. Since communication is a two-way thing, 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 seprate 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.