Network and Distributed Systems Laboratory Session

Session 5

Development of a basic client and server over TCP in Java. The server will provide a basic text capitalisation service.

Task 1: Server Specification (class ServerTCP.java)

The server specifications:

- The port on which you will receive requests will be passed as an argument on the command line. For example: java `ServerTCP 12345`
- Once a connection to a client has been established, it waits to receive text to be modified (read it with `readLine`) from the connected socket.
- The server's reply will depend on the request:
  - If it receives `END` it will return `OK` and close the connection.
  - Otherwise it will return the text capitalised.
- Once the connection is closed, the server will again wait for a new connection and service request.
- The server shall report its current status on the standard output (screen).
- The server must be able to recover (correctly release resources and re-host another client) if the client terminates the connection incorrectly.
- The server can only have one client waiting (**the queue of pending clients must be 1**).

Task 2: Client Specification (class ClientTCP.java)

Customer specifications:

- The IP address and port of the server to which the client should connect will be passed as an argument on the command line. For example:
  `java ClientTCP 192.168.1.2 12345`
- Once the client is connected, it should continuously request the user for the text to capitalise (a line of text) and send it to the server (using `println`).
- After each sending of the text to be processed, the client must wait for the response from the server containing the modified text.
- When the user wants to terminate, he/she will type with the value `TERMINATE.`
- When the client detects that the user wants to terminate it will send the text `END`, wait for the server's response (`OK`) and close the connection.
- During the whole execution the client must inform the user (by typing on the screen) its status (e.g.: `Connected to 192.168.1.2:12345, Waiting for response...`).
- If the client sends data and the connection is found to be closed, it **closes the client in an orderly manner**.

Task 3: Trace capture

Simulate the following behaviours by taking the generated traffic trace with Wireshark. If the client and server are on the same machine, use the `loopback` IP (`127.0.0.1`) as the IP of the server. As interface to capture you should use the interface called `Adapter for loopback traffic capture`.

TCP1 behaviour (trace 1 - **p5e1-4.pcapng**):

- Start the server and then the client.
- In the client send a single message and then type `TERMINATE` to end it.

TCP2 behaviour (trace 2 - **p5e5.pcapng**):

- Without having any server active, try to start the client.

TCP3 behaviour (trace 3 - **p5e6-7 .pcapng**):

- Start the server.
- Then start 3 clients trying to connect to that server.
- Write `TERMINATE` on clients who have successfully logged in to finalise shipments.

Network and Distributed Systems Laboratory Session

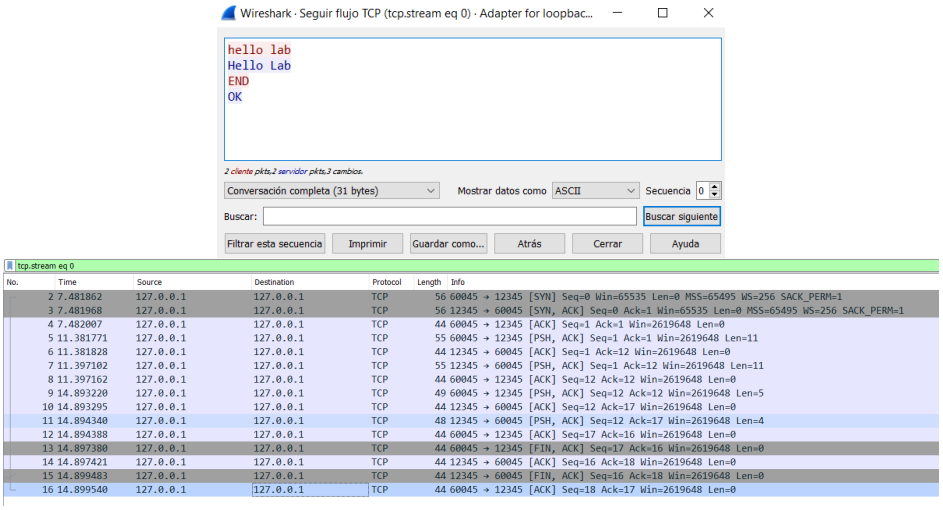
Task 4: Analysis of our TCP protocol

Answer the following questions using the traces captured above.

Using the TCP1 trace (**p5e1-4.pcapng**):

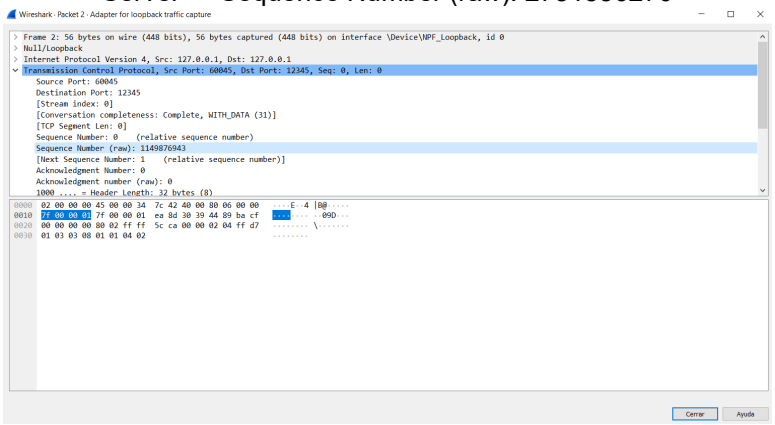
Exercise 1. Identify a frame of the communication and use the "`Follow TCP stream`" option to see the information exchange between client and server. Which port is the client using? And the server?

Server: 12345
Client: 60455



Exercise 2. What is the number of the sequence number (absolute) used by the TCP client to the server? And the responses from the server to the client?

Client -> Sequence Number (raw): 1149876943
Server -> Sequence Number (raw): 2781886270



Exercise 3. Specify the segments related to the following activities and which `Socket` and `ServerSocket` methods are responsible for the exchange of these segments:

a) Initialisation of the connection.
b) Sending data.
(c) Close of the connection.

2 7.481862	127.0.0.1	127.0.0.1	TCP	56 60045 → 12345 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
3 7.481968	127.0.0.1	127.0.0.1	TCP	56 12345 → 60045 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
4 7.482007	127.0.0.1	127.0.0.1	TCP	44 60045 → 12345 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
5 11.381771	127.0.0.1	127.0.0.1	TCP	55 60045 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=11
6 11.381828	127.0.0.1	127.0.0.1	TCP	44 12345 → 60045 [ACK] Seq=1 Ack=12 Win=2619648 Len=0
7 11.397102	127.0.0.1	127.0.0.1	TCP	55 12345 → 60045 [PSH, ACK] Seq=1 Ack=12 Win=2619648 Len=11
8 11.397162	127.0.0.1	127.0.0.1	TCP	44 60045 → 12345 [ACK] Seq=12 Ack=12 Win=2619648 Len=0
9 14.893220	127.0.0.1	127.0.0.1	TCP	49 60045 → 12345 [PSH, ACK] Seq=12 Ack=12 Win=2619648 Len=5
10 14.893295	127.0.0.1	127.0.0.1	TCP	44 12345 → 60045 [ACK] Seq=12 Ack=17 Win=2619648 Len=0
11 14.894340	127.0.0.1	127.0.0.1	TCP	48 12345 → 60045 [PSH, ACK] Seq=12 Ack=17 Win=2619648 Len=4
12 14.894388	127.0.0.1	127.0.0.1	TCP	44 60045 → 12345 [ACK] Seq=17 Ack=16 Win=2619648 Len=0
13 14.897380	127.0.0.1	127.0.0.1	TCP	44 60045 → 12345 [FIN, ACK] Seq=17 Ack=16 Win=2619648 Len=0
14 14.897421	127.0.0.1	127.0.0.1	TCP	44 12345 → 60045 [ACK] Seq=16 Ack=18 Win=2619648 Len=0
15 14.899483	127.0.0.1	127.0.0.1	TCP	44 12345 → 60045 [FIN, ACK] Seq=16 Ack=18 Win=2619648 Len=0
16 14.899540	127.0.0.1	127.0.0.1	TCP	44 60045 → 12345 [ACK] Seq=18 Ack=17 Win=2619648 Len=0

Gray initialization, blue exchange of messages; gray again, end of the connection.

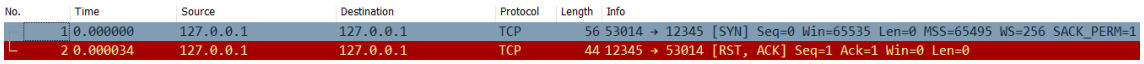
Network and Distributed Systems Laboratory Session

Exercise 4. How many sequence numbers are consumed on each side (client and server) during connection initiation and closure?

Initialization 1 Server 1 client.
Closure 1 Server 1 client. Unless we count END/OK as the closure and have three bytes and 2 to transmit more.
Due to the difference of the difference od END-OK and flags message

Using the TCP2 trace (**p5e5 .pcapng**):

Exercise 5. Does the client's connection attempt receive any kind of reply? If so, does it have any special characteristics?



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	56	53014 → 12345 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
2	0.000034	127.0.0.1	127.0.0.1	TCP	44	12345 → 53014 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

RST

Using the TCP3 trace (**p5e6-7.pcapng**):

Exercise 6. Are all 3 clients able to connect? If any of them failed to connect, is there any indication that the queue is full?

No, only the first one (connected) and the second (queue); the third receives RST answer.

Exercise 7. Are waiting clients (i.e. those in the queue) initialised or is this initialisation done when they are freed from the queue (with the `accept` method)?

The clients stop at the readline from their code and won't be released until the server releases its current client.

Network and Distributed Systems Laboratory Session

Server

```
import java.io.*;
import java.net.*;

class ServerTCP {
	public static String capitalize(String s) {
		String words[] = s.split("\\s");
		String res = "";
		for (String w : words) {
			if (!res.isEmpty()) {
				res += " ";
			}
			res += w.substring(0, 1).toUpperCase() + w.substring(1);
		}
		return res;
	}

	public static void main(String[] args) {
		// SERVER DATA
		// * FIXED: If read from command line must be commented out
		// int port = 12345; // server port
		// * VARIABLE: If read from command line it must be uncommented
		int port = Integer.parseInt(args[0]); // 12345

		// SOCKETS
		ServerSocket server = null; // Passive (receiving requests)

		Socket client = null; // Active (client service)

		// FLOWS FOR SENDING AND RECEIVING
		BufferedReader in = null;
		PrintWriter out = null;

		// Create and initialise server socket (passive socket)
		try {
			server = new ServerSocket(port, 1); // Queue Size = 1 / Backlog
		} catch (IOException e) {
			System.err.println("Error: cannot connnect to port " + port);
			System.exit(1);
		}

		while (true) // Loop to receive incoming connections
		{
			// Waiting for incoming connections
			System.out.println("STATUS: Waiting for clients");
			try {
				client = server.accept(); // Blocks until a connection is binded
			} catch (IOException e) {
				System.err.println("Error: could not accept a client on port:" +
port);
				System.exit(1);
			}

			SocketAddress clientAddress = client.getRemoteSocketAddress();

			System.out.println("STATUS: Client " + clientAddress + "accepted.");
			// Once a connection is accepted, initialise input/output streams of the
			// connected socket

			try {
				out = new PrintWriter(client.getOutputStream(), true);
				in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
			} catch (IOException e) {
				System.err.println("Error: could not get I/O for  client " +
clientAddress);
				System.exit(1);
			}

			System.out.println("STATUS: Client connected from " + clientAddress);

			boolean exit = false;
			while (!exit) // Start loop of a client service
```

```
                        {
                                String line = null;
                                // Receive inline text sent by the client via the input stream
of the
                                // connected socket
                                try {
                                        line = in.readLine();
                                } catch (IOException e) {
                                        System.err.println("Error: Could not get the input from
client " + clientAddress);
                                        System.exit(1);
                                }
                                System.out.println("STATUS: Received " + line + " from the
client");
                                // Check if it is the end of connection - REPLACE WITH 'END'
STATEMENT
                                if (line.compareTo("END") != 0) {
                                        line = capitalize(line);

                                        // Send text to the client via the outgoing stream of
the connected
                                        // socket
                                        out.println(line);
                                        System.out.println("STATUS: Sending to the client: " +
line);
                                } else { // Client wants to close connection, has sent END
                                        exit = true;
                                        out.println("OK");
                                }
                        } // end of service
                        System.out.println("STATUS: Closing the connection with " +
clientAddress);
                        // Close flows and socket
                        try {
                                in.close();
                                out.close();
                                client.close();
                        } catch (IOException e) {
                                System.err.println("Error: Closing the connection");
                                System.exit(1);
                        }

                        System.out.println("STATUS: Connection closed");
                } // end of loop
        } // end of method
}
```

Network and Distributed Systems Laboratory Session

Client

```
import java.io.*;
import java.net.*;

public class ClientTCP {

	public static void main(String[] args) throws IOException {
		// SERVER DATA:
		// * FIXED: comment them out if you read them from the command line.
		// String serverName = "127.0.0.1"; // local address
		// int serverPort = 12345;
		// * VARIABLES: uncomment them if you read them from the command line
		String serverName = args[0];
		int serverPort = Integer.parseInt(args[1]);

		// SOCKET
		Socket serviceSocket = null;

		// FLOWS FOR SENDING AND RECEIVING
		PrintWriter out = null;
		BufferedReader in = null;

		// * TODO: Create socket and connect to server
		try {
			System.out.println("STATUS: Connecting to the server");
			serviceSocket = new Socket(serverName, serverPort);
		} catch (IOException e) {
			System.err.println("Error: cannot connnect to " + serverName);
			System.exit(1);
		}
		// Initialise input/output streams of connected socket in PrintWriter
		// and BufferedReader variables
		try {
			out = new PrintWriter(serviceSocket.getOutputStream(), true);
			in = new BufferedReader(new
InputStreamReader(serviceSocket.getInputStream()));
		} catch (IOException e) {
			System.err.println("Error: could not get I/O for  " + serverName);
			System.exit(1);
		}

		// Receive welcome message from server and display it
		System.out.println("STATUS: Connected to the server");
		// Get text from keyboard
		BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
		String userInput;

		System.out.println("Enter a text to be sent (TERMINATE to finish)");
		userInput = stdIn.readLine();

		// * TODO: Check if the user has started the end of the interaction
		while (userInput.compareTo("TERMINATE") != 0) { // service loop
			// Send text in userInput to the server via the output stream of the
			// connected socket
			out.println(userInput);
			System.out.println("STATUS: Sending " + userInput);
			System.out.println("STATUS: Waiting for a reply");
			// Receive text sent by the server via the input stream of the connected
socket
			String line = null;
			try {
				line = in.readLine(); // If it is waiting at the queue this is
the last line waiting
			} catch (IOException e) {
				System.err.println("Error: could not read the message of server
" + serverName);
				System.exit(1);
			}

			System.out.println("Reply received: " + line);

			// Read user text by keyboard
			System.out.println("Enter a text to be sent (TERMINATE to finish)");
```

```
                        userInput = stdIn.readLine();
                } // End of client service loop

                System.out.println("STATUS: Closing the connection to the server");
                // We exit because the client wants to end the interaction, it has entered
                // TERMINATE.
                // Send END to the server to indicate the end of the service.
                out.println("END");
                // * TODO: Receive OK from Server
                String ok = in.readLine();
                System.out.println("STATUS: Closing . . .");
                // Close flows and socket
                out.close();
                in.close();
                stdIn.close();
                serviceSocket.close();
                System.out.println("STATUS: Connection closed");
        }
}
```