Practice 4

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

The model client /service is defined by the permanently active service of the server and the request of them from the client processes. For sending/receiving messages we use sockets.

A **socket** is an endpoint (source or destination) for communication between processes running on different computers. The socket interface is an API to realise distributed applications over a network. We can choose a TCP or UDPs socket.

The number of ports go from 0 to 1023 well-known, 1025 to 49.151 registered and 49.152 to 65.535 private.

Can be done with the java.net package:

- **Programming with TCP sockets in Java** we need the class **ServerSocket** is used for connection-oriented servers; whereas **Socket** for a connection-oriented C/S connection. We will use a buffer for reading and writing
- **Programming with UDP sockets in Java** by creating **DatagramSocket** objects and loop the sending/receiving in server and client.

To handle the IP addresss we use InetAddress, getByName/getLocalHost.

MulticastSocket class is used for multicast.

Network and Distributed Systems Laboratory Session

Task 1 (Professor-guided): Client Specification (ClientUDP.java class)

Customer specifications:

- The IP address and port of the server to which the client should send to will be passed as an argument on the command line. For example: `java UDPclient 192.168.1.2 12345`
- Once the client is connected, it must continuously request the user for a (one line) text to be sent to the server.
- After each sending, the client must wait for the reply from the server containing the modified text.
- When the user wants to finish, he/she will type the text `TERMINATE` on the keyboard.
- When the client detects that the user wants to terminate, it will terminate the programme (without sending anything).
- 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…`).

Task 2: Server Specification (ServerUDP.java class)

The server specifications:

- The port on which you will receive requests will be passed as an argument on the command line. For example: `java UDPserver 12345`
- The server will be constantly waiting for datagrams that include as data a text string to be modified.
- Use a 400 Byte buffer for reception.
- The server will send a reply to the client from which it received the datagram.
- The server's reply will be the capitalised text.
- The server shall report its current status on the standard output (screen).

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 server IP. As interface to capture you should use the interface called `Adapter for loopback traffic capture`.

UDP1 behaviour (trace 1 - **p4e1-3.pcapng**):

- Start the server and then two clients.
- Send multiple messages on each client and alternately.
- In one of the clients send a message with accent marks.
- Finally type `TERMINATE` on both to terminate the clients.

UDP2 behaviour (trace 2 - **p4e4.pcapng**):

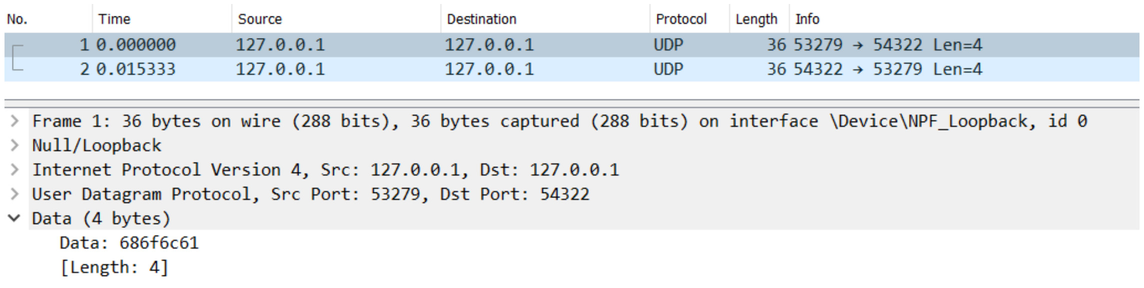
- Without having any server active, try to start the client.
- Send a message to the server and subsequently stop (interrupt) the execution of the client.

Network and Distributed Systems Laboratory Session

Task 4: Analysis of our UDP protocol

Answer the following questions using the traces captured above.

Using the UDP1 capture (**p4e1-3.pcapng**):



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	36	53279 → 54322 Len=4
2	0.015333	127.0.0.1	127.0.0.1	UDP	36	54322 → 53279 Len=4

```
> Frame 1: 36 bytes on wire (288 bits), 36 bytes captured (288 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 53279, Dst Port: 54322
v Data (4 bytes)
    Data: 686f6c61
    [Length: 4]
```

Exercise 1. Which is the port used by the client, which is the port used by the server, and what types of ports are?

Client is >49511 (ephemeras), server is 54322 (private).

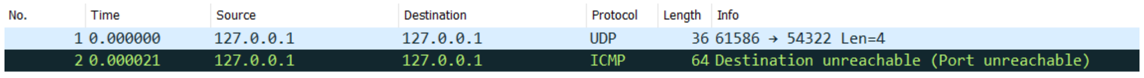
Exercise 2. Wireshark offers the option to "Follow UDP stream", but in UDP there is no such concept. How is Wireshark able to decide whether a message belongs to one "stream" or another?

With the client and user port.

Exercise 3. Examine a message with accent marks in it, does the size indicated in the length field of the UDP header match the number of letters sent? Why?

The length is increased as accents occupies more bits on UTF-8.

Using the UDP2 capture (**p4e4.pcapng**):



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	36	61586 → 54322 Len=4
2	0.000021	127.0.0.1	127.0.0.1	ICMP	64	Destination unreachable (Port unreachable)

Exercise 4. Why do you manage to send if there is no active server? Do you receive any response? If yes, please indicate what this response means and whether it is processed or not.

You can connect to the port, but receives no answer as there is no socket interface it interacts with.

Without using Wireshark captures:

Exercise 5. Make sure you catch the UDP socket creation exception and display (`getMessage()` method) the error that occurs (modify the code if it did not). Try to open the server twice with the same parameters, what error does it indicate that occurs? What should you do to fix this error and have two servers of the same type on your machine?

Two servers connected to the same port, avoiding so would solve it.

Network and Distributed Systems Laboratory Session

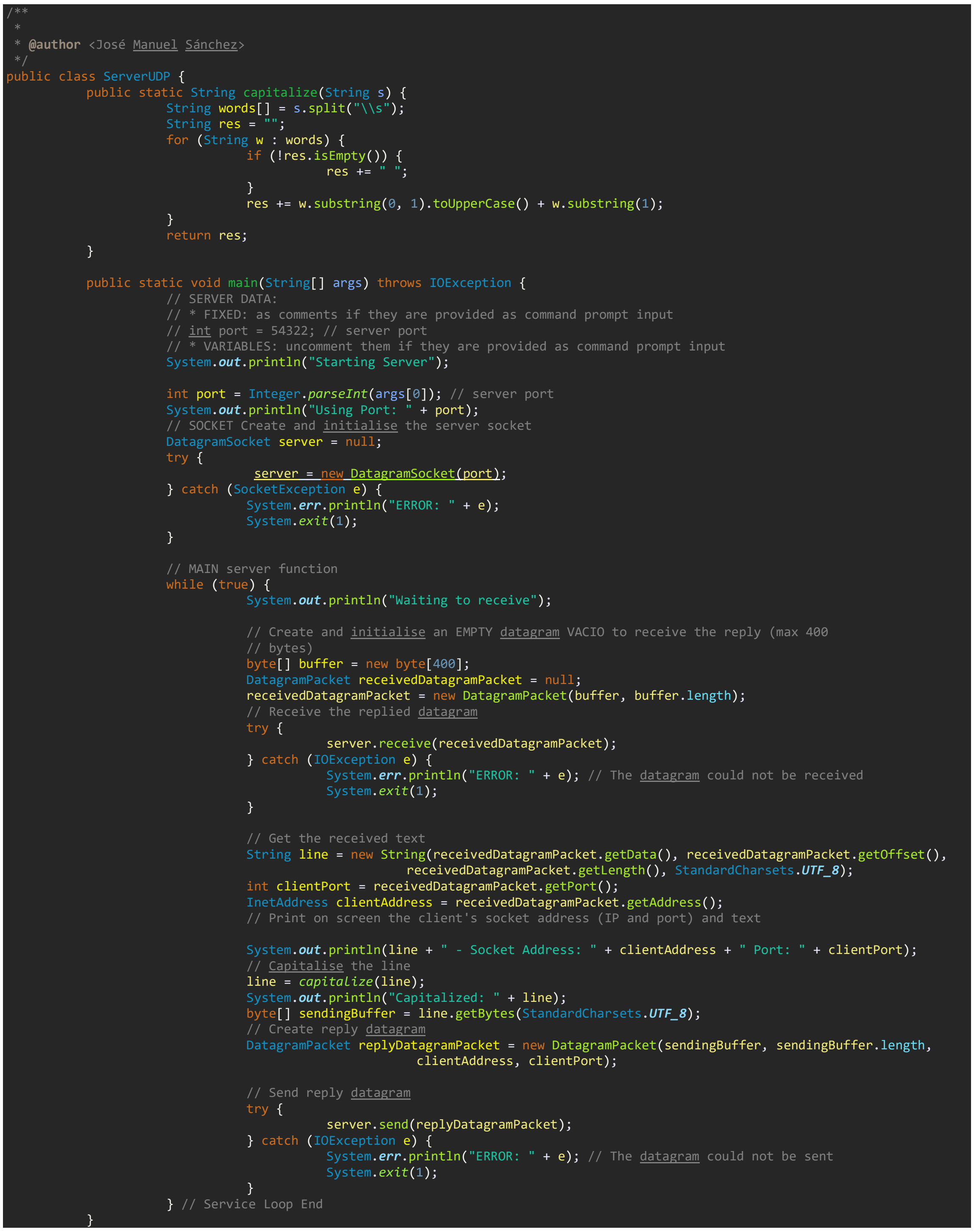
CODE

CLIENT

```java
/**
 * 
 * @author <José Manuel Sánchez>
 */

public class ClientUDP {
	public static void main(String[] args) throws IOException {
		// SERVER DATA:
		// * FIXED: as comments if they are provided as command prompt input
		// String serverName = "127.0.0.1"; //localhost
		// int serverPort = 54322;
		// * VARIABLES: uncomment them if they are provided as command prompt input
		String serverName = args[0]; // "127.0.0.1"
		int serverPort = Integer.parseInt(args[1]);

		// Create socket
		DatagramSocket serviceSocket = null;
		try {
			serviceSocket = new DatagramSocket();
		} catch (SocketException e) {
			System.err.println("ERROR: " + e);
			System.exit(1);
		}

		// INITIALISE KEYBOARD INPUT
		BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
		String userInput;
		System.out.println("Type a text to send (TERMINATE to finish): ");
		userInput = stdIn.readLine(); /* Saved the string in userInput */

		// Check if the user wants to TERMINATE the service
		while (userInput.compareTo("TERMINATE") != 0) {
			// * Create a datagram with the given String
			DatagramPacket dp = null;
			try {
				dp = new DatagramPacket(userInput.getBytes(StandardCharsets.UTF_8),
						userInput.getBytes(StandardCharsets.UTF_8).length,
InetAddress.getByName(serverName),
						serverPort);
				// It's important to codify with the right standard to avoid length and strange
				// characters errors.

			} catch (UnknownHostException e) {
				System.err.println("ERROR: " + e); // Server not running.
				System.exit(1);
			}

			// Send the datagram through the socket
			try {
				serviceSocket.send(dp);
			} catch (IOException e) {
				System.err.println("ERROR: " + e); // The datagram could not be sent
				System.exit(1);
			}

			System.out.println("STATUS: Waiting for the reply");

			// Create and initialise an EMPTY datagram to receive the reply (max 400
			// bytes)
			byte[] buffer = new byte[400];
			DatagramPacket receivedDatagramPacket = new DatagramPacket(buffer, buffer.length);
			// Receive the replied datagram
			try {
				serviceSocket.receive(receivedDatagramPacket);
			} catch (IOException e) {
				System.err.println("ERROR: " + e); // The datagram could not be received
				System.exit(1);
			}
			// Extract the content of the datagram body in variable line
			String line = new String(receivedDatagramPacket.getData(), receivedDatagramPacket.getOffset(),
					receivedDatagramPacket.getLength(), StandardCharsets.UTF_8);
			System.out.println("echo: " + line);

			System.out.println("Type a text to send (TERMINATE to finish): ");
			userInput = stdIn.readLine();
		}

		System.out.println("STATUS: Closing client");

		serviceSocket.close();

		System.out.println("STATUS: closed");
	}
}
```

Network and Distributed Systems Laboratory Session

SERVER



```java
/**
 *
 * @author <José Manuel Sánchez>
 */
public class ServerUDP {
	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) throws IOException {
		// SERVER DATA:
		// * FIXED: as comments if they are provided as command prompt input
		// int port = 54322; // server port
		// * VARIABLES: uncomment them if they are provided as command prompt input
		System.out.println("Starting Server");

		int port = Integer.parseInt(args[0]); // server port
		System.out.println("Using Port: " + port);
		// SOCKET Create and initialise the server socket
		DatagramSocket server = null;
		try {
			server = new DatagramSocket(port);
		} catch (SocketException e) {
			System.err.println("ERROR: " + e);
			System.exit(1);
		}

		// MAIN server function
		while (true) {
			System.out.println("Waiting to receive");

			// Create and initialise an EMPTY datagram VACIO to receive the reply (max 400
			// bytes)
			byte[] buffer = new byte[400];
			DatagramPacket receivedDatagramPacket = null;
			receivedDatagramPacket = new DatagramPacket(buffer, buffer.length);
			// Receive the replied datagram
			try {
				server.receive(receivedDatagramPacket);
			} catch (IOException e) {
				System.err.println("ERROR: " + e); // The datagram could not be received
				System.exit(1);
			}

			// Get the received text
			String line = new String(receivedDatagramPacket.getData(), receivedDatagramPacket.getOffset(),
					receivedDatagramPacket.getLength(), StandardCharsets.UTF_8);
			int clientPort = receivedDatagramPacket.getPort();
			InetAddress clientAddress = receivedDatagramPacket.getAddress();
			// Print on screen the client's socket address (IP and port) and text

			System.out.println(line + " - Socket Address: " + clientAddress + " Port: " + clientPort);
			// Capitalise the line
			line = capitalize(line);
			System.out.println("Capitalized: " + line);
			byte[] sendingBuffer = line.getBytes(StandardCharsets.UTF_8);
			// Create reply datagram
			DatagramPacket replyDatagramPacket = new DatagramPacket(sendingBuffer, sendingBuffer.length,
					 clientAddress, clientPort);

			// Send reply datagram
			try {
				server.send(replyDatagramPacket);
			} catch (IOException e) {
				System.err.println("ERROR: " + e); // The datagram could not be sent
				System.exit(1);
			}
		} // Service Loop End
	}
```

Bibliography

Distributed Programming: Sockets in Java Lecturer: Daniel Muñoz