# *SWitCH - Computing systems and networks (SCOMRED)*

2018/2019

Laboratory Script - Week 05

IMPERIOUS: After each practical activity (in yellow), students are supposed to reflect on results, and wait for the teacher's acknowledgment before progressing to the next practical activity in this script.

## 1. Java UDP client and server

We will start our Java sockets adventure by implementing and testing a pair of simple UDP client and a server demo applications.

The client reads a text line from the user's terminal and sends it to the server inside a UDP datagram, then it waits for a response, a UDP datagram as well, containing a text line to be presented at the terminal.

The mission for the server application is receiving a text line, **mirroring it**, and then send it back to the client as response.

The server application is not supposed to exit ever, the client application exits if the user types exit as text line to be sent.

Source files **UdpCli.java** and **UdpSrv.java** are available on Moodle (Samples folder).

Take a while to analyse line by line the provided code in the following pages.

## 1.1. **UdpCli.java**

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

class UdpCli {
       static InetAddress serverIP;

       public static void main(String args[]) throws Exception {
              byte[] data = new byte[300];
              String phrase;

              if(args.length!=1) {
                     System.out.println("Server IP address/DNS name is required as argument");
                     System.exit(1);
                     }

              try { serverIP = InetAddress.getByName(args[0]); }
              catch(UnknownHostException ex) {
                     System.out.println("Invalid server address supplied: "  + args[0]);
                     System.exit(1);
                     }

              DatagramSocket sock = new DatagramSocket();
              DatagramPacket udpPacket = new DatagramPacket(data, data.length, serverIP, 9999);

              BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

              while(true) {
                     System.out.print("Sentence to send (\"exit\" to quit): ");
                     phrase = in.readLine();
                     if(phrase.compareTo("exit")==0) break;
                     udpPacket.setData(phrase.getBytes());
                     udpPacket.setLength(phrase.length());
                     sock.send(udpPacket);
                     udpPacket.setData(data);
                     udpPacket.setLength(data.length);
                     sock.receive(udpPacket);
                     phrase = new String( udpPacket.getData(), 0, udpPacket.getLength());
                     System.out.println("Received reply: " + phrase);
                     }
              sock.close();
              }
       }
```

- The getByName() method of the InetAddress class is used to transform the command line argument string representing the server (IP address or DNS name) into an InetAddress object holding the server's IP address.

- A DatagramSocket class object is created, the used constructor takes no arguments, so any local free port number will be assigned.

- A DatagramPacket object is created, the used constructor receives the content of the datagram (payload), the payload's number of bytes, the destination IP address, and the destination port number. This DatagramPacket object will be used for both sending the request and receiving the reply.

- A line of text is read from the console to a string, if the content is exit then the loop is finished and the application exists after closing the socket.

- Otherwise, the string content is set as payload for the datagram and the datagram can then be sent by calling the socket's send() method.

- The DatagramPacket object is then prepared for receiving the server reply, the buffer and maximum datagram size are set. Next a reply datagram can be received by calling the socket's receive() method. This is a blocking operation.

- When (and if) a reply UDP datagram arrives, receive() method unblocks filling data in the DatagramPacket object and the buffer.

- For the purpose of printing at the console, a string is created from the received datagram payload.

## 1.2. **UdpCli.java**

```
import java.net.*;

class UdpSrv {
	static DatagramSocket sock;
	public static void main(String args[]) throws Exception {
		byte[] data = new byte[300];
		byte[] data1 = new byte[300];
		int i, len;

		try { sock = new DatagramSocket(9999); }
		catch(BindException ex) {
			System.out.println("Bind to local port failed");
			System.exit(1);
			}

		DatagramPacket udpPacket= new DatagramPacket(data, data.length);
		System.out.println("Listening for UDP requests (IPv6/IPv4). CTRL+C to terminate");
		while(true) {
			udpPacket.setData(data);
			 udpPacket.setLength(data.length);
			 sock.receive(udpPacket);
			 len=udpPacket.getLength();
			 System.out.println("Request from: " +
				udpPacket.getAddress().getHostAddress() + " port: " +
				udpPacket.getPort());
			 for(i=0;i<len;i++) data1[len-1-i]=data[i];
			 udpPacket.setData(data1);
			 udpPacket.setLength(len);
			 sock.send(udpPacket);
			}
		}
	}
```

- A DatagramSocket class object is created, the constructor used receives an integer fixed local port number to bind the socket to. This, of course may, raise an exception if that UDP port number is already being used on the local host.

- A new DatagramPacket object is created, the constructor used defines only a buffer and the buffer size. The IP address and port number are set when a datagram is received (source IP address and source port number).

- The server then starts a never ending loop for receiving requests and sending corresponding replies. The receive() method is called to receive the datagram (client request), if no datagram has been received, the thread will block until one arrives.

- After receiving the request the source IP address and source port number are stored in the DatagramPacket object, so there is no need to change them when sending a reply because the same DatagramPacket is used for that purpose. Source IP address and source port number are printed at the server's console.

- A mirrored version of the string carried by the request UDP datagram is created and defined as the payload of the reply datagram. The reply is sent by calling the socket's send() method.

PRACTICE:

Source files **UdpCli.java** and **UdpSrv.java** are available on Moodle (Samples folder), assuming you are using an IDE (Integrated Development Environment), create to standard Java Application projects (console applications), named **UdpCli** and **UdpSrv** with default main classes.

Each project should have now a source file with the project's name and a class with the same name defining the **main()** method. Replace those files contents with the source code available at Moodle, except for the package declaration at the top.

On the **UdpCli** project you must also define the command line argument, check your project settings. This is required because this application uses the first command line argument to establish the server's IP address, it can be either an IPv4 address, an IPv6 address or a DNS name. For now your client is going to use the server you will have running on your laptop, so set it to **127.0.0.1** (or localhost).

Now let's test, for now locally only:

1st - Start the server (**UdpSrv**) and let it running.

2nd - Start the client (**UdpCli**)

At the client's console you should see the prompt:

**Sentence to send ("exit" to quit):**

Enter some text line, if everything went as expected, the sentence was sent by **UdpCli** to your local **UdpSrv**, which mirrored it and then sent it back to **UdpCli** to print it.

Send some more text lines, check at **UdpSrv** console for log messages it should be printing for each received request.

When done type exit at the **UdpCli** console. You may let **UdpSrv** running.

Now let's test again, but now with the server on a remote node:

Because wireless networks enforce some restrictions regarding traffic between nodes associated to the same access point, you should connect to a DEI VPN service before proceeding.

Ask a nearby colleague for the IP node address he is using.

(He can get that at site https://rede.dei.isep.ipp.pt/myip)

Also make sure he is running **UdpSrv** on his laptop and that is letting traffic go through his firewall.

On your **UdpCli** project's settings change the command line argument to match your colleague's IP address.

Now run **UdpCli** again and test it.

It should work as before, yet now it's your colleague's **UdpSrv** processing and replying to your requests, your colleague should be able to see that on his **UdpSrv** console.

## 2. Java TCP chat client and server

Now we are going to start a more complex project to build a TCP based Chat service, the concept is rather simple.

There is a TCP server (**TcpChatSrv**) to which client applications (**TcpChatCli**) may connect (establish TCP connections), then clients can send text lines which the server is obliged to deliver to every connected client.

Text lines sent by clients should have a nickname prefix enclosed in brackets, but the server ignores that, is simply forwards the entire line to all connected clients.

Some things must be established regarding transactions between the clients and the server, this is usually called the application protocol.

- One issue we have to handle in TCP is how does the receiver (reader) knows how many bytes it should read, we are talking about variable length text lines so this must be addressed. The proposed solution is the sender (writer) first sends a single byte representing the line length in number of characters, and then send the line itself.
- Another point is how a client ends it session, it could simply close the TCP connection, that would generate an error on the server side and it would know the client has ended its session. Nevertheless, it's better to have a smoother way to end the session, in the proposed solution the client informs it wants to end the session by sending an empty line (simply sending the byte value zero). In this case the server should also reply back with the zero byte.

### 2.1. The client

The client application establishes a TCP connection with the server, thus it will have only one connected socket, despite that it must handle two possible input events:

- A message sent by the server application, it should print to the user.
- A message typed by the user it should send to the server.

Either of these two events may happen at any time, so we have an asynchronous events issue and we will handle that by using threads. The initial thread will have the mission of reading lines from the user's terminal and sending them to the server through the TCP connection, but before that, another thread is started to receive and print messages sent by the server through the TCP connection.

### 2.1.1. TcpChatCli.java

Contains the **main()** method to start the client application and it will also start a thread defined in TcpChatCliConn.java to handle incoming messages from the server.

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

class TcpChatCli {
    static InetAddress serverIP;
    static Socket sock;
    static private final int SERVER_PORT=9999;


    public static void main(String args[]) throws Exception {
        String nick, phrase;
        String server="127.0.0.1";
        byte[] data = new byte[300];

        if(args.length!=1) {
            System.out.println(
                    "No server IPv4/IPv6 address or DNS name provided on command line, using 127.0.0.1");
        } else {
            server=args[0];
        }

        try { serverIP = InetAddress.getByName(server); }
        catch(UnknownHostException ex) {
            System.out.println("Invalid server: " + server);
            System.exit(1); }

        try { sock = new Socket(serverIP, SERVER_PORT); }
        catch(IOException ex) {
            System.out.println("Failed to connect to server " + server + ":" + SERVER_PORT);
            System.exit(1); }

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        DataOutputStream sOut = new DataOutputStream(sock.getOutputStream());

        System.out.println("Connected to server " + server + ":" + SERVER_PORT);
        System.out.print("Enter your nickname: "); nick = in.readLine();

        // start a thread to read incoming messages from the server
        Thread serverConn = new Thread(new TcpChatCliConn(sock));
        serverConn.start();

        // read messages from the console and send them to the server
        while(true) {
            phrase=in.readLine();
            if(phrase.compareTo("exit")==0)
            { sOut.write(0); break;}
            phrase="(" + nick + ") " + phrase;
            data = phrase.getBytes();
            sOut.write((byte)phrase.length());
            sOut.write(data,0,(byte)phrase.length());
        }

        serverConn.join(); // wait for the connection termination on the thread
        sock.close();
    }
}
```

- If no server is provider in the command line, the localhost (127.0.0.1) is used as default.
- Once the TCP connection is established, a nickname is prompted, it will be used as prefix on every message sent by the client. Then a thread is started to handle incoming messages from the server.
- The loop reads a line from the user's terminal if is **exit** it simply sends the zero byte telling the server it wants to end the session, it will then wait for the thread to terminate (join).
- Otherwise the message is sent to the server, first one byte with the length a then the message itself (with the nickname prefix).

### 2.1.2. TcpChatCliConn.java

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

    class TcpChatCliConn implements Runnable {
        private Socket s;
        private DataInputStream sIn;

        public TcpChatCliConn(Socket tcp_s) { s=tcp_s;}

        public void run() {
            int nChars;
            byte[] data = new byte[300];
            String phrase;

            try {
                sIn = new DataInputStream(s.getInputStream());
                while(true) {
                    nChars=sIn.read();
                    if(nChars==0) break;
                    sIn.read(data,0,nChars);
                    phrase = new String(data, 0, nChars);
                    System.out.println(phrase);
                }
            }
            catch(IOException ex) { System.out.println("I/O error: client disconnected."); }
        }
    }
```

- This thread starts by reading one byte incoming from the server, if zero then it's a confirmation of the server to our own request to end the session, so we terminate the thread's execution (and the join() method in the TcpChatCli class will unblock).
- If the received byte is not zero then is the message length and we read it and print it on the user's terminal.

## 2.2. The server

As with any TCP server we must use threads to handle each client connection, but in this specific implementation the server also has to manage a list of connected clients, when it receives a text line from one client it will have to write it back to every connected client, including the one who sent it.

When a client ends its session or the TCP connection is broken, it must be removed from the list of connected clients.

One additional feature to implement is providing to new clients, just connected, the 25 most recent messages so the user can get into the context of the conversation.

To have a more flexible implementation, the TCP server itself will be a thread which will then accept TCP connections from clients and start new threads for each.

### 2.2.1. TcpChatSrv.java

```
public class TcpChatSrv {
    private static final int TCP_SERVER_PORT=9999;

    public static void main(String args[]) throws Exception {

        Thread tcpServer=new TcpServer(TCP_SERVER_PORT);
        tcpServer.start();

        tcpServer.join();
    }
}
```

- This class simply starts the TcpServer thread and waits for it to end (something that will never happen).
- By implementing the main() method this way, we could for instance start several TCP servers on different port numbers.

### 2.2.2. TcpServer.java

The TcpServer class is a Thread's subclass, it creates a ServerSocket on the provided port number and then starts accepting client's connections.

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

public class TcpServer extends Thread {

    private int serverPort;

    public TcpServer(int port) {serverPort=port;}

    @Override
    public void run() {
        ServerSocket sock;
        Socket cliSock;
        try { sock = new ServerSocket(serverPort); }
        catch(IOException ex) {
            System.out.println("TCP server: local port number " + serverPort +
                                               " not available - server aborted");
            return;
        }
        System.out.println("TCP server: ready, listening on local port number " + serverPort);
        while(true) {

            try { cliSock=sock.accept(); }
            catch(IOException ex) {
                System.out.println("TCP server: failed to accept client connection");
                cliSock=null;
            }
            if(cliSock!=null) {
                Thread cliThread = new TcpServerConn(cliSock);
                cliThread.start();
                System.out.println("TCP server: new client connection from " +
                     cliSock.getInetAddress().getHostAddress() +
                     ", port number " + cliSock.getPort());

            }
        }
    }
}
```

- Several logging messages are printed at the terminal.
- When a connection is accepted the connected socket is provided to the constructor of the TcpServerConn class, itself a thread that is next started.

### 2.2.3. TcpServerConn.java

The TcpServerConn class has two missions:

- The static part is used to manage the connected clients list, it will be referenced by the class name.
- The non-static part is used to create threads, one for each client to read incoming messages from them.

Starting by the static part of the class:

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

public class TcpServerConn extends Thread {

    // Class stuff (static)

    private final static String tcpServerConnIOError =
                       "TCP server: I/O Error. Client connection aborted";

    // List of connected TCP clients
    private static HashMap<Socket,DataOutputStream> cliList = new HashMap<>();

    public static synchronized void sendToAll(int len, byte[] data) throws Exception {
        for(DataOutputStream cOut: cliList.values()) {
            cOut.write(len);
            cOut.write(data,0,len);
        }
    }

    public static synchronized void sendLastMessagesToCli(Socket s) throws Exception {
        ChatMessages.sendLast(cliList.get(s),false);
    }


    public static synchronized void addCli(Socket s) throws Exception {
        cliList.put(s,new DataOutputStream(s.getOutputStream()));
    }

    public static synchronized void remCli(Socket s) throws Exception {
        DataOutputStream cOut = cliList.get(s);
        cliList.remove(s);
        cOut.write(0);
        s.close();
    }


    // Instance stuff
```

- To store connected clients, a HashMap is used, the access key is the client's connect socket, the stored object the corresponding DataOutputStream.
- All methods accessing the client's list are synchronized (to the class).
- The sendToAll() method sends a provided message to all connected clients in the list, it gets all OutputStreams of all clients in the list and writes the message on all of them (first the length and then the message itself).
- The sendLastMessagesToCli() method calls a static method from another class, yet to be defined, dedicated to storing the last 25 transmitted messages.

The non-static part of the class is used by the TcpServer class to create instances and then start running them as threads.

```
        // Instance stuff

        Socket cli;
        private DataInputStream sIn;

        public TcpServerConn(Socket s) { cli=s; }

        @Override
        public void run() {
            int nChars;
            byte[] data = new byte[300];

            try {
                sIn = new DataInputStream(cli.getInputStream());
                addCli(cli);
                sendLastMessagesToCli(cli);
                while(true) {
                    nChars=sIn.read();
                    if(nChars==0) break; // an empty line means the client wants to exit
                    sIn.read(data,0,nChars);
                    sendToAll(nChars,data);
                    ChatMessages.push(new String(data,0,nChars));
                }
            }
            catch(Exception ex) { System.out.println(tcpServerConnIOError); }
            System.out.println("TCP server: client " + cli.getInetAddress().getHostAddress() +
                    ", port number " + cli.getPort()+ " disconnected");
            try { TcpServerConn.remCli(cli); } catch(Exception ex) {
    System.out.println(tcpServerConnIOError); }

        }
    }
```

- The constructor simply stores the connected socket for later use by the run() method when the thread is started.
- The run() method starts by adding the new client to the clients list (addCli() method) and then calling the sendLasMessagesToCli() method to send the last 25 messages to it.
- The loop runs until the client sends the s zero byte (session end request) or the connection is broken (IOException). When either thing happens the client is removed from the list.
- For each received message from the client, the sendToAll() method is called to deliver that message to every connected client. Also, the push() method of the yet to define class is used to place it in the list of 25 last messages.

### 2.2.4. ChatMessages.java

The ChatMessages class is totally static and has the sole mission of storing the last 25 sent messages. Messages are stored in an array of strings and two public methods are available, they are being used by the previous class.

```
import java.io.*;

public class ChatMessages {

    private static final int MAX_MESSAGES=25;

    private static String[] lastMessages = new String[MAX_MESSAGES];
    private static int numMessages=0;

    public static synchronized void sendLast(DataOutputStream cOut, boolean lastFirst) throws IOException {
        if(lastFirst) {
            for(int i=0;i<numMessages;i++) {
                cOut.write(lastMessages[i].length());
                cOut.write(lastMessages[i].getBytes(),0,lastMessages[i].length());
            }
        } else {
            for(int i=numMessages-1;i>=0;i--) {
                cOut.write(lastMessages[i].length());
                cOut.write(lastMessages[i].getBytes(),0,lastMessages[i].length());
            }
        }
    }

    public static synchronized void push(String msg) {
        for(int i=numMessages;i>0;i--) {
            if(i<MAX_MESSAGES) {
                lastMessages[i]=lastMessages[i-1];
            }
        }
        lastMessages[0]=msg;
        if(numMessages<MAX_MESSAGES) numMessages++;
    }
}
```

- The sendLast() method writes stored messages through a provided OutputStream, in the sequence specified by the second Boolean argument.
- The push() method adds a message to the array.

PRACTICE:

Create to empty projects for Java standard console applications, named **TcpChatCli** and **TcpChatSrv**.

Source code files are available at Moodle (Samples folder).

**TcpChatCli** project: add the two files (**TcpChatCli.java** and **TcpChatCliConn.java**) to the project and define the **TcpChatCli.java** as the class holding the **main()** method to run the application.

**TcpChatSrv** project: add the four files (**TcpChatSrv.java**, **TcpServer.java**, **TcpServerConn.java**, and **ChatMessages.java**) to the project and define the **TcpChatSrv.java** as the class holding the **main()** method to run the application.

TESTING 1:

1st - Start by running the server (**TcpChatSrv**) on your laptop, the server prints several logging messages on the terminal, so later come back to it to check them.

2nd - Run one first client on your laptop, by default (no command line arguments), it connects the server on the localhost, so it should be now connected to your local server, you may check that on the server's console. Proceed with the client use defining a nickname and then sending some text lines.

3rd - Run additional clients on your laptop, use different nicknames on each to make things clearer. Check the server's console for logging messages.

4th - Try ending sessions, the smooth way (by typing exit) and the hazardous way by simply stopping the client application and thus broking the TCP connection.

TESTING 2:

1st - Settle a team of nearby partners to share the same server, one team member will run the server and share with others the IP address his laptop is using.

Notes regarding wireless restrictions apply here as well. To remove this possible obstacle team members can connect to a DEI VPN service. Also the team member providing the service should allow incoming traffic for the server application running.

2nd - All team members are now going to connect to the same server, on the **TcpChatCli** project's runtime definitions they should settle the first command line argument of be the IP address of the team member providing the service.

3rd - All team members may now run the client, they should use different nicknames to avoid confusion. As you have noticed, unique nicknames are not enforced by these applications.