UNIVERSITATEA TEHNICĂ A MOLDOVEI

FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

PROTOCOLUL TCP

LUCRARE DE LABORATOR NR. 2

la disciplina ”Programarea în rețea”

Autor:

studentul gr. TI 141,

învățământ cu frecvența redusă

MIROVSCHI Eugen

Profesor:

Lector universitar,

ANTOHI Ionel

_____

(semnătura)

Chișinău, 2018

I. Scopul lucrării

Studierea protocolului TCP și utilizarea acestuia pentru a comunica cu un server la distanță.

II. Sarcina lucrării

De dezvoltat 2 aplicații: Client și Server, ce vor avea posibilitatea de comunicare.

Clientul va cunoaște adresa serverului și va crea o conexiune către acesta prin declararea unui socket. Acest socket va fi folosind pentru trimiterea și recepționarea mesajelor.

Serverul va avea un punct terminal de conexiune care va indica spre orice adresă, însă portul trebuie s fie unic și identic cu cel specificat în client. Aplicația server, spre deosebire de client, pe lângă recepționarea și transmiterea mesajelor, trebuie să accepte și să gestioneze conexiunile clienților. Pentru aceasta se folosește un socket specific numit ServerSocket.

Instrucțiunile de conexiune, transmitere și recepționare pot eșua, respectiv erorile acestora trebuie prelucrate folosind try-catch. Metodele de transmitere și recepționarea lucrează cu fluxuri de octeți, respectiv este necesar convertirea mesajelor în șiruri de octeți. Citirea fluxului poate fi efectuată folosind un ciclu while atâta timp cât conexiunea e deschisă.

III. Efectuarea lucrării

Clasele folosite în lucrarea precedentă oferă posibilitatea de a comunica cu un server la un nivel înalt folosind un protocol definit. Acest impune un anumit mod de lucru, limitând posibilitățile de a comunica în rețea. Astfel platforma Java oferă un set de instrumente pentru a crea un model de comunicare server-client, folosind interfețe de un nivel mai jos.

În aplicațiile client-server, de obicei serverul oferă careva servicii cum ar fi executarea unor cereri în baza de date. Clientul folosește aceste servicii cum ar fi salvarea datelor în baza de date sau citirea acestora pentru a le afișa la ecran.

Pentru comunicare între server și client vor fi folosite socket-uri. Un socket este un punct final dintr-o conexiune bidirecțională între două programe care rulează în noduri diferite în cadrul rețelei. Socket-urile sunt folosite pentru a reprezenta conexiunea între client și server. Implicit, java oferă 2 clase de acest tip numite Socket și ServerSocket, ambele sunt locate în pachetul java.net.

Pentru ca un client să inițializeze o conexiune către server, trebuie să creeze un socket care se va accesa serverul știind-ui adresa IP și portul. Exemplu de creare:

```
socket = new Socket(serverIp, serverPort);
```

În caz că serverul acceptă această conexiune, aplicația va continua să lucreze în continuare conform algoritmului stabilit, în caz contrar, aceasta va arunca o excepție care trebuie prelucrată folosind blocul try-catch.

După deschiderea conexiunii putem comunica cu serverul folosind fluxurile de intrare și ieșire.

Serverul, pe de altă parte, nu doar trimite și recepționează mesaje. Acesta trebuie să asculte și să gestioneze conexiunile clienților. Pentru a efectua acest lucru vom folosi clasa ServerSocket. Exemplu de inițializare și ascultare:

```
private void internalRun() throws IOException
{
    serverSocket = new ServerSocket(port, MAX_CONNECTIONS);
    System.out.println("Server listening on port " + port);

    while (!serverSocket.isClosed())
    {
        accept();
    }
}

private void accept()
{
    try
    {
        final Socket socket = serverSocket.accept();
        System.out.println("Client connected " + socket.toString());

        try
        {
            executorService.execute(() -> handleClient(socket));
        }
        catch (final RejectedExecutionException e)
        {
            System.out.println("Server full, rejecting:" + socket);
            commandHandler.write(socket.getOutputStream(), new
ConnectResponseCommand(false));
        }
    }
    catch (final IOException e)
    {
        System.out.println("Client connection failed:" + e.getMessage());
    }
}

private void handleClient(final Socket socket)
{
    try (final ServerHandler serverHandler = serverHandlerFactory.apply(socket))
    {
```

```
            serverHandler.run();
        }
        catch (final IOException e)
        {
            System.out.println("An error occurred during while handling client [" + socket
+"]: " + e.getMessage());
        }
    }
```

Pentru fiecare client, este creat un fir de execuție aparte care se ocupa de gestionarea acestuia.

Spre diferență de UDP, protocolul TCP transmite datele doar prin fluxuri și nu oferă conceptul de mesaj. Respectiv rămâne la discreția noastră să adăugăm acest concept prin folosirea unui delimitator. Această operațiune, împreună cu convertirea mesajelor pot fi abstractizate într-o clasă aparte cum ar fi CommandHandler:

```
public class DelimiterCommandHandler implements CommandHandler
{
    private static final int COMMAND_DELIMITER = 1;
    private static final String COMMAND_FIELD_DELIMITER = "\u0002";
    private static final String COMMAND_DATA_DELIMITER = "\u0003";
    private static final String COMMAND_DATA_VALUE_DELIMITER = "\u0004";

    private final CommandFactory commandFactory;

    public DelimiterCommandHandler(final CommandFactory commandFactory)
    {
        this.commandFactory = commandFactory;
    }

    @Override
    public <T extends Command> T read(final InputStream inputStream) throws IOException
    {
        final String command = readUntil(inputStream, COMMAND_DELIMITER);
        final String[] commandFields = command.split(COMMAND_FIELD_DELIMITER);

        final String commandType = commandFields[0];
        final Map<String, String> data =
Stream.of(commandFields[1].split(COMMAND_DATA_DELIMITER))
            .map(row -> row.split(COMMAND_DATA_VALUE_DELIMITER))
            .collect(toMap(row -> row[0], row -> row[1]));

        return commandFactory.createCommand(commandType, data);
    }

    private String readUntil(final InputStream inputStream, final int end) throws
IOException
    {
        int data;
        final List<Byte> bytes = new LinkedList<>();

        while ((data = inputStream.read()) != end && data != -1)
        {
            bytes.add((byte) data);
        }

        if (data == -1)
        {
            throw new IOException("Client disconnected");
        }
```

```
        final byte[] bytesArray = new byte[bytes.size()];
        IntStream.range(0, bytes.size()).forEach(i -> bytesArray[i] = bytes.remove(0));

        return new String(bytesArray);
    }

    @Override
    public void write(final OutputStream outputStream, final Command command) throws
IOException
    {
        outputStream.write(command.getCommandType().getBytes());
        outputStream.write(COMMAND_FIELD_DELIMITER.getBytes());
        outputStream.write(serialize(command.getData()).getBytes());
        outputStream.write(COMMAND_DELIMITER);
        outputStream.flush();
    }

    private String serialize(final Map<String, String> data)
    {
        return data.entrySet().stream()
            .map(entry -> entry.getKey() + COMMAND_DATA_VALUE_DELIMITER +
entry.getValue())
            .collect(joining(COMMAND_DATA_DELIMITER));
    }
}
```

Pentru simplificare algoritmului a fost citirea este efectuată câte un byte. Deși se recomandă folosirea unui buffer, pentru a optimiza performanță, această schimbare complica modul de citire deoarece buffer-ul poate avea informație din 2 comenzi diferite.

Un alt element folosit pentru simplificarea comunicării între client și server a fost adăugarea conceptului de comenzi (mesaje). Acestea reprezintă niște clase simple care păstrează un anumit set de informații. Exemplu de comenzi create: ConnectResponseCommand, RegisterCommand, RegisterResponseCommand, SendMessageCommand, RecieveMessageCommand etc. Acestea pot fi serializate și deserealizate pentru a putea fi scrise și citite în cadrul unui flux de date.

Pentru a efectua această lucrare am decis să aleg o tematică și anume un chat simplu între mai mulți clienți. Pentru ca utilizatorul să se conecteze la chat, trebuie să-și introducă numele care va fi unic în cadrul serverului și va fi folosit ca identificator. Exemplu de fereastră pentru înregistrare este reprezentat în figura 1.

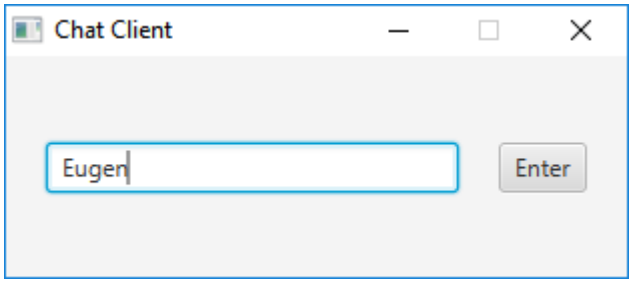

Figura 1 – Exemplu de înregistrare

În caz că serverul este plin sau numele este deja ocupat, va fi afișat un mesaj de eroare ca în figura 2.

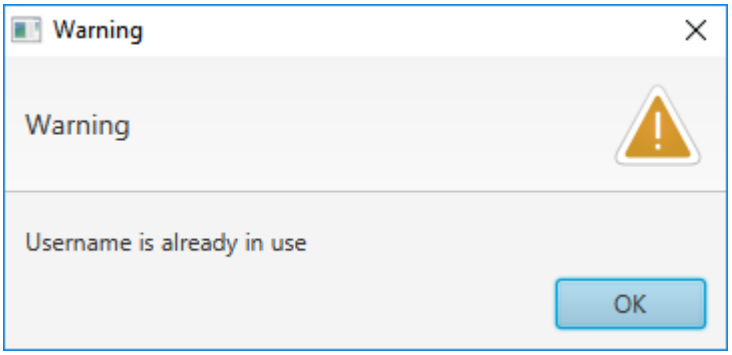

Figura 2 – Exemplu de eroare

După autentificarea cu succes, se va deschide fereastra chat unde utilizatorul va putea comunica cu alți clienți:

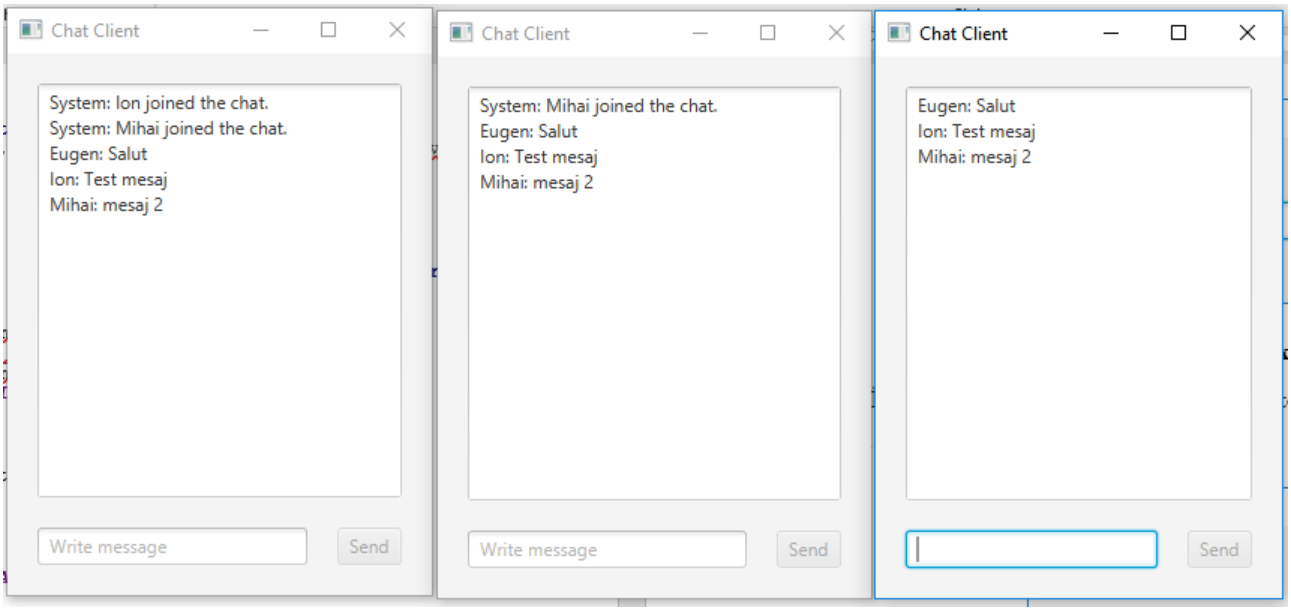

Figura 3 – Fereastra chat

IV. Concluzii

Efectuând această lucrare de laborator am studiat conceptele de bază a protocolului TCP și modul de utilizare pentru a crea un canal de comunicare între o aplicație de tip server și alta de tip client.

Platforma Java oferă implicit instrumente necesare pentru comunicare între două aplicații folosind protocolul TCP. În comparație cu instrumentele studiate în cadrul efectuării lucrării precedente, acestea au o interfață definită la un nivel mai jos. Astfel ne permită să decidem noi modelul de comunicare cum ar fi structura mesajelor, ordinea de primire și transmitere a acestora etc.

Pentru ca o comunicare între două aplicații să aibă loc, acestea trebuie să definească așa numitele Socket-uri care reprezintă punctele final ale canalului de comunicare.

Pentru a crea un Socket de tip client, este necesar să cunoaștem adresa serverului și portul de comunicare. În caz ca conexiunea are loc cu succes program va continua să execute comenzile definite cum ar fi transmiterea și citirea datelor. În caz contrar va fi aruncată o excepție care trebuie prelucrată folosind blocul try-catch.

Spre diferență de client, serverul are ca responsabilitate recepționarea conexiunilor și gestionarea acestora. Pentru aceasta se folosește un socket specific numit ServerSocket. Atunci când un client se conectează, ServerSocket creează o instanță nouă de Socket simplu care ulterior va fi folosit pentru comunicarea cu acel client.

Protocolul TCP transmite datele folosind fluxuri și nu definește exact ce reprezintă un mesaj. Respectiv devine responsabilitatea noastră să creăm modelul de comunicare folosind aceste fluxuri de date. În cadrul acestei lucrări am decis să separ mesajele folosind un delimitator unic.