

Ministerul Educației al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Inginerie Software Automatică

Raport

Lucrarea de laborator nr. 1

Disciplina: Programarea produselor program

Tema: Agent de mesaje

A realizat

Stratan I.

St. gr. TI-144

A verificat

Pecari M.

Lect. sup.

Chișinău 2017

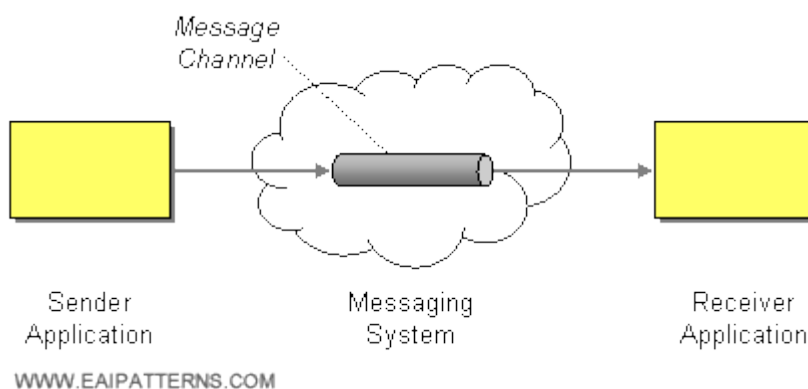
Obiective

- Studiul agenților de mesagerie;
- Elaborarea unui protocol de comunicare al agentului de mesaje;
- Tratarea concurentă a mesajelor;
- Alegerea protocolului de transport (în dependență de scopul/domeniul de aplicare al agentului de mesaje);
- Alerea și elaborarea strategiei de păstrare a mesajelor;

Sarcina de bază

- Elaborarea protocolului de comunicare. Descrie protocolul într-un fișier markdown și salvează-l în mapa „docs” din repositoryul tău;
- Alegerea protocolului de transport și argumentarea alegerii;
- Implementarea cozii de mesaje (utilizând colecții de date concurente)

Implementează o coadă de mesaje care poate avea atât multipli producători (expeditori) de mesaje, cât și multipli consumatori de mesaje.



Sistemul trebuie să permită:

- plasarea unui mesaj în coadă (concurent de mai mulți producători);
- consumarea mesajelor (concurent de mai mulți consumatori);

Efectuarea lucrări:

Acest model se bazează pe protocolul TCP. Într-o aplicație rețea întotdeauna avem două părți: o parte **client** care inițializează conversația și trimite cereri, și o parte **server** care primește cererile și răspunde la acestea. Clientul întotdeauna creează un soclu pentru a iniția conversația și trebuie să cunoască serverul cărui adresă trimite cererea, iar serverul trebuie să fie pregătit pentru a recepționa aceste cereri. În momentul recepționării mesajului creează un soclu pe partea serverului, soclu care va facilita deservirea clientului. Atât pe partea de client cât și pe partea de server se utilizează câte un obiect de tip *Socket* pentru comunicare. Pe partea de server mai trebuie să cream un obiect de tip *ServerSocket*, care are sarcina primirii conexiunilor și acceptarea acestora.

```

public class SocketServer implements Runnable {
    private final Map<Socket, ServerClient> socketClientMap = new
ConcurrentHashMap<>();
    @Override
    public void run() {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(8888);
            while(true){
                Socket socket = serverSocket.accept();
                ServerClient client = new ServerClient(socket);
                socketClientMap.put(socket,client);
                new Thread(client).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Clientul trebuie sa cunoasca doua lucruri despre server: numele serverului (utilizat pentru determinarea adresei IP al serverului) si numarul portului la care acesta asculta cererile clientilor.

```

public class Client implements Runnable {

    @Override
    public void run() {
        int serverPort = 8888;
        String address = "127.0.0.1";
        QueueManager queueManager = new QueueManager();
        try {
            InetAddress ipAddress = InetAddress.getByName(address);
            System.out.println("Any of you heard of a socket with IP address " +
address + " and port " + serverPort + "?");

            Socket socket = new Socket(ipAddress, serverPort);
            System.out.println("Conectat <3");

            InputStream sin = socket.getInputStream();
            OutputStream sout = socket.getOutputStream();

            DataInputStream in = new DataInputStream(sin);
            DataOutputStream out = new DataOutputStream(sout);

            BufferedReader keyboard = new BufferedReader( new
InputStreamReader(System.in));
            String line = null;

            new Thread(()->{

                String response;
                try {
                    while ((response=in.readUTF())!=null) {
                        System.out.println("primit message:" + response);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }

            }).start();
        }
    }
}

```

```

        while ((line=keyboard.readLine())!=null) {

            System.out.println("Seding this line to server...");
            out.writeUTF(line);
            out.flush();

        }
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
}

```

Serverul concurent permite deservirea in paralel a mai multor clienti. Aceasta paralelitate se poate realiza prin crearea a mai multor procese fiu, cate unul pentru fiecare client sau prin crearea de fire de executie pentru deservirea fiecarui clienti

```

public class ServerClient implements Runnable {

    private final Socket socket;

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

    @Override
    public void run() {
        try {
            InputStream sin = socket.getInputStream();
            OutputStream sout = socket.getOutputStream();

            DataInputStream in = new DataInputStream(sin);
            DataOutputStream out = new DataOutputStream(sout);

            String line = null;

            while((line=in.readUTF())!=null) {

                System.out.println("primit message:" + line);
                Message message = MessageParser.parse(line);
                QueueManager queueManager = new QueueManager();

                if(message.getCommand().equals("put")) {
                    System.out.println("inseram:" + message.getPayload());
                    queueManager.addMessage(message.getPayload());
                }
                else if(message.getCommand().equals("get")) {

                    String messagePayload=queueManager.getMessageQueue();

                    out.writeUTF(messagePayload);
                    out.flush();

                    System.out.println("returnam:" + messagePayload);

                }

            }

        }
    }
}

```

```

    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

```

Pentru a stoca mesajele clientului, trebuie de creat o coadă de mesaje, în care dacă clientul utilizează comanda GET, atunci face o cerere la coadă pentru a returna toate mesajele, iar dacă utilizează comanda PUT, atunci introduce mesajul în coadă. Pentru aceasta s-a creat următoarele clase:

```

public class QueueManager {
    private static Queue messageQueue = new LinkedBlockingQueue() ;

    public String getMessageQueue() {
        return (String) messageQueue.poll();
    }
    public void addMessage(String message){
        messageQueue.add(message);
    }
}

```

Această clasă de mai jos este folosită pentru a delimita mesajul în două părți:

1. Command
2. Payload

```

public class MessageParser {
    public static Message parse(String message){

        String []splitedMessage=message.split(",");
        Message ms = new Message() ;
        if(splitedMessage.length>0)
            ms.setCommand(splitedMessage[0]);

        if(splitedMessage.length>1)
            ms.setPayload(splitedMessage[1]);

        return ms;
    }
}

```

Următoarea clasă este Mesajul, care are următoarele metode getPayload, setPayload, getCommand și setCommand.

```

public class Message {
    private String command;
    private String payload;

    public String getPayload(){
        return payload;
    }
}

```

```

    }
    public void setPayload(String payload) {
        this.payload = payload;
    }

    public String getCommand() {
        return command;
    }

    public void setCommand(String command) {
        this.command = command;
    }
}

```

Această clasă este folosită atunci cînd clientul folosește comanda GET pentru a returna toate mesajele din coadă și PUT pentru a adăuga mesajul în coadă.

```

public class CommandMessageManager {
    QueueManager queueManager = new QueueManager();
    Message message = new Message();

    public void execute(Message message) {
        if (message == null) {
            return;
        }

        if (message.getCommand().equalsIgnoreCase("get")) {
            queueManager.getMessageQueue();
        }
        else if (message.getCommand().equalsIgnoreCase("put")) {
            queueManager.addMessage(message.getPayload());
        }
    }
}

```

Pentru a rula serverul este utilizată următoarele instrucțiuni:

```

public static void main(String[] args) {
    SocketServer socketServer = new SocketServer();
    Thread socketThread = new Thread(socketServer);
    socketThread.start();
}

```

Iar pentru a rula clientul se folosește instrucțiunile de mai jos:

```

public static void main(String[] args)
{
    Client client = new Client();
    Thread clientThread = new Thread(client);
    clientThread.start();
}

```

Verificare

În imaginea de mai jos (figura 1) este afișat rularea serverului.

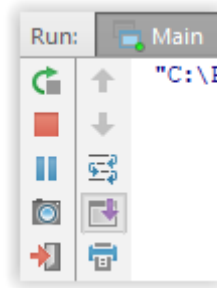


Figura 1 – Rularea serverului

În (figura 2) este afișat rularea clientului care este conectat la IP adresa 127.0.0.1 și la portul 8888.

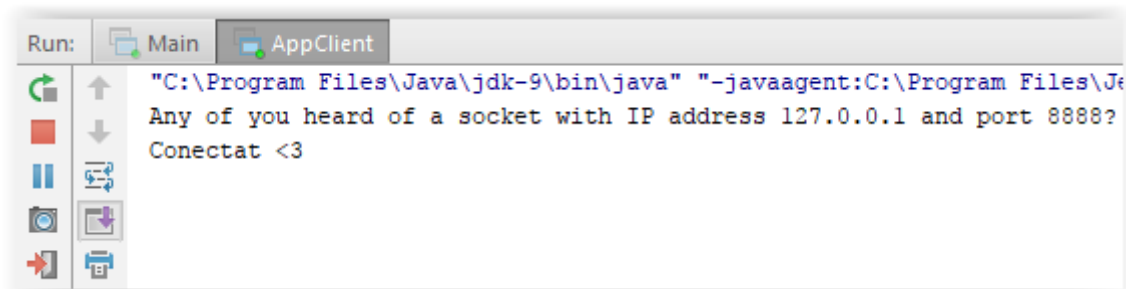


Figura 2 – Rularea clientului

Clientul efectuează inserarea unui mesaj în coadă și apoi returnează mesajul din coadă, această posibilitate este realizată în imaginea următoare (figura 3).

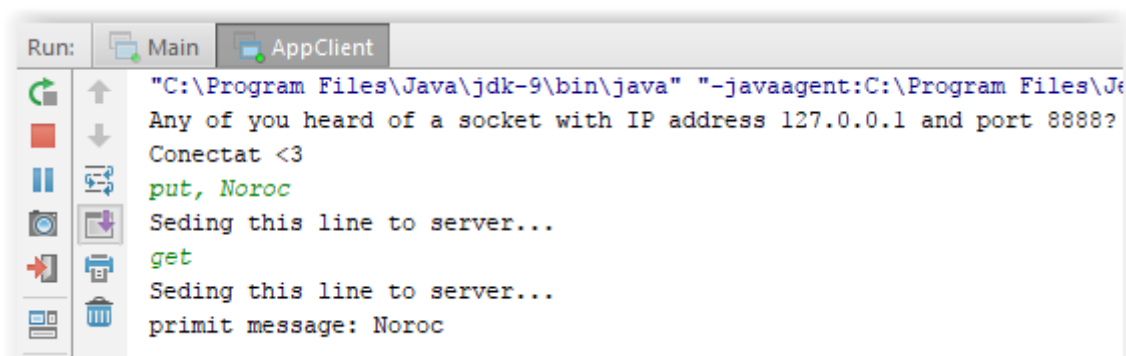
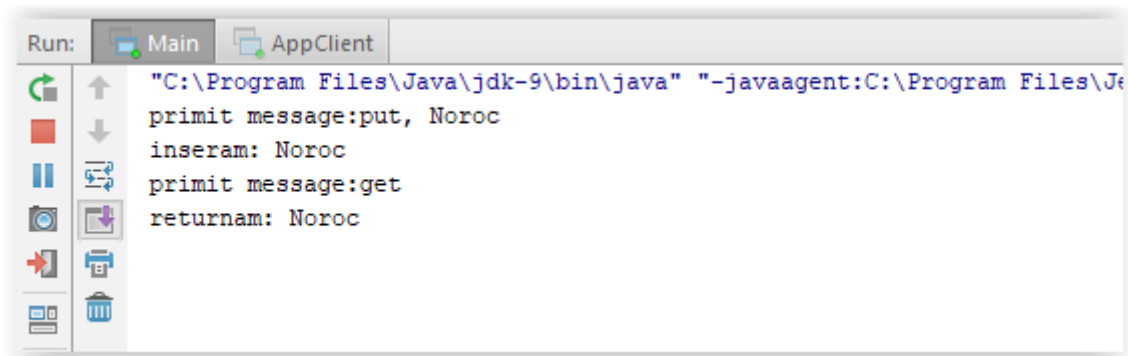


Figura 3 – Clientul utilizează comenzile PUT și GET

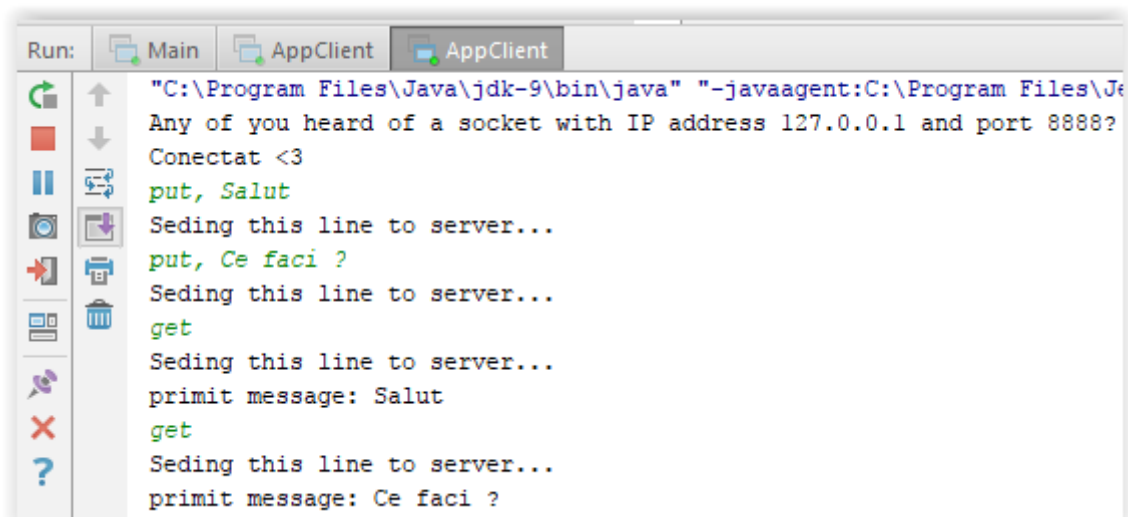
Mai jos este demonstrat stocarea mesajelor pe server (figura 4).



```
Run: Main
"C:\Program Files\Java\jdk-9\bin\java" "-javaagent:C:\Program Files\Java\jdk-9\bin\javaagent.jar"
primit message:put, Noroc
inseram: Noroc
primit message:get
returnam: Noroc
```

Figura 4 – Stocarea mesajelor pe server

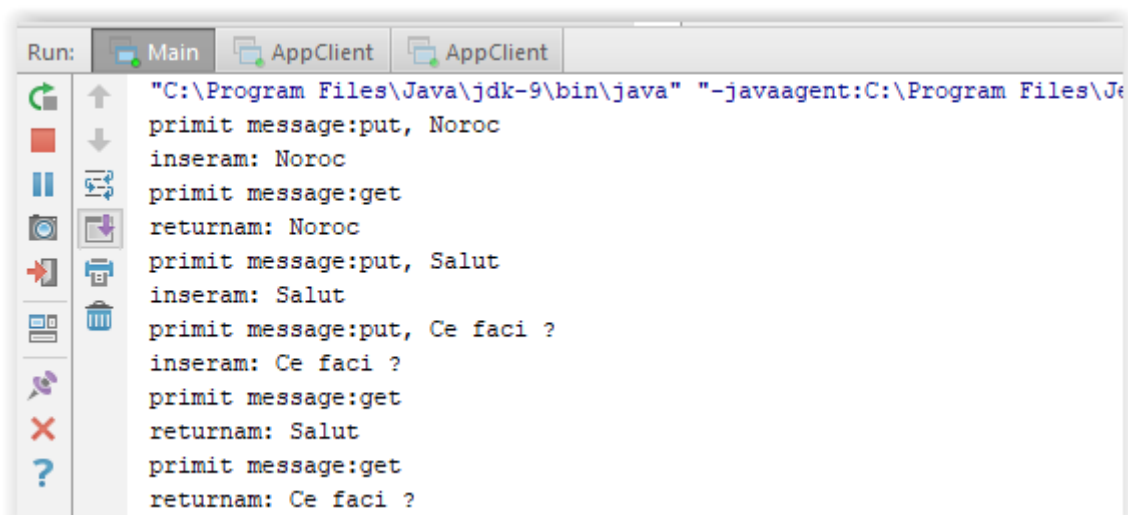
Încercăm să conectăm un nou client la server, după care introducem careva mesaje și apoi le returnăm (figura 5).



```
Run: Main AppClient
"C:\Program Files\Java\jdk-9\bin\java" "-javaagent:C:\Program Files\Java\jdk-9\bin\javaagent.jar"
Any of you heard of a socket with IP address 127.0.0.1 and port 8888?
Conectat <3
put, Salut
Seding this line to server...
put, Ce faci ?
Seding this line to server...
get
Seding this line to server...
primit message: Salut
get
Seding this line to server...
primit message: Ce faci ?
```

Figura 5 – Conectarea unui nou client la server

În imaginea de mai jos (figura 6) urmărim starea lui.



```
Run: Main AppClient
"C:\Program Files\Java\jdk-9\bin\java" "-javaagent:C:\Program Files\Java\jdk-9\bin\javaagent.jar"
primit message:put, Noroc
inseram: Noroc
primit message:get
returnam: Noroc
primit message:put, Salut
inseram: Salut
primit message:put, Ce faci ?
inseram: Ce faci ?
primit message:get
returnam: Salut
primit message:get
returnam: Ce faci ?
```

Figura 6 – Starea serverului

Concluzii

Efectuarea acestui laborator, mi-a dat interesul de a învăța să creez aplicații distribuite, deoarece în prezent toate aplicațiile au conexiune la internet și folosesc protocoalele de comunicație. În acest laborator am folosit protocolul TCP/IP care reprezintă cel mai flexibil protocol de transport disponibil și permite computerelor din întreaga lume să comunice între ele, indiferent de tipul sistemului de operare ce rulează pe ele. TCP/IP oferă un foarte mare grad de corecție al erorilor, deși nu este ușor de utilizat și nici cel mai rapid protocol. Printre avantajele utilizării acestui protocol se numără: este un protocol de rețea routabil suportat de majoritatea sistemelor de operare. Reprezintă o tehnologie pentru conectarea sistemelor diferite. Se pot utiliza mai multe utilitare de conectivitate standard pentru a accesa și transfera date între sisteme diferite. Este un cadru de lucru robust, scalabil între platforme client / server. Reprezintă o metodă de acces la resursele interne.