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 aplicațiilor distribuite

Tema: Agent de mesaje

A realizat:
St. gr. TI-142 D. Gorduz

A verificat:

Lect. sup. M. Pecari

Chişinău 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);

- Alegerea și elaborarea strategiei de păstrare a mesajelor;

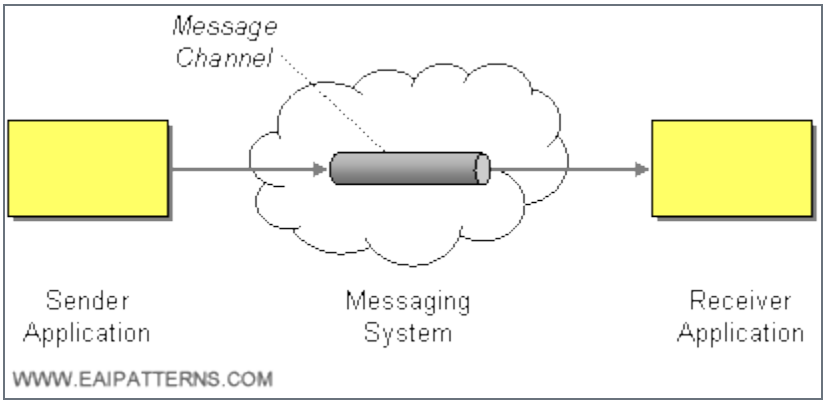
Sarcina de bază

- Elaborarea protocolului de comunicare. Descrie protocolul într-un fișier markdown și salvează-l în mapa „docs” din repozitoriul 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);

Mersul lucrării

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 căruia adresează cererea, iar serverul trebuie să fie pregătit pentru a recepționa aceste cereri. În momentul recepționării mesajului creează un soclu pe partea serverului, soclu care vă 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 creăm 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";
        try {
            InetAddress ipAddress = InetAddress.getByName(address);
```

```
            Socket socket = new Socket(ipAddress, serverPort);
            System.out.println("Connection established <@>");

            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("Message received:" + response);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }).start();

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

                System.out.println("Sending 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 clienți.

```
public class ServerClient implements Runnable {

    private final Socket socket;

    private final QueueManager queueManager = QueueManager.getInstance();

    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("Message received:" + line);
                Message message = MessageParser.parse(line);

                if(message.getCommand().equalsIgnoreCase("put")){
                    System.out.println("<@>Inserting:" + message.getPayload());
                    queueManager.addMessage(message.getPayload());
                }
                else if(message.getCommand().equalsIgnoreCase("get")){

                    String  messagePayload=queueManager.getMessageQueue();

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

                    System.out.println("<@>Returning:" + 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 QueueManager() { }

    private static final QueueManager _INSTANCE = new QueueManager();

    private final BlockingQueue<String> messageQueue = new ArrayBlockingQueue<String>(10) ;

    public static synchronized QueueManager getInstance() {
        return _INSTANCE;
    }

    public String getMessageQueue() {
        return 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;
    }
}
```

Rularea aplicației

Se rulează doi clienți concomintent și un server, conform imaginii de mai jos:

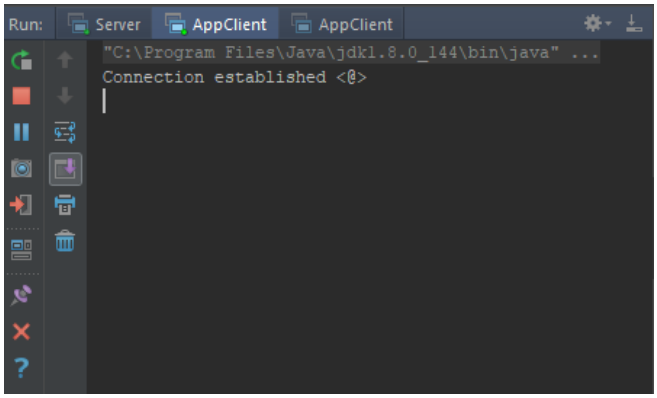


Figura 1 – Inițierea serverului și clienților

Se inserează mesajele în primul client și observăm că serverul primește mesajul(figura 2) și îl stochează în coadă(figura 3). Când dorim să luăm mesajul din coadă, se face prin comanda *get* și se returnează primul element din coadă și apoi se șterge, apoi cel de-al doilea devine primul ș.a.m.d.

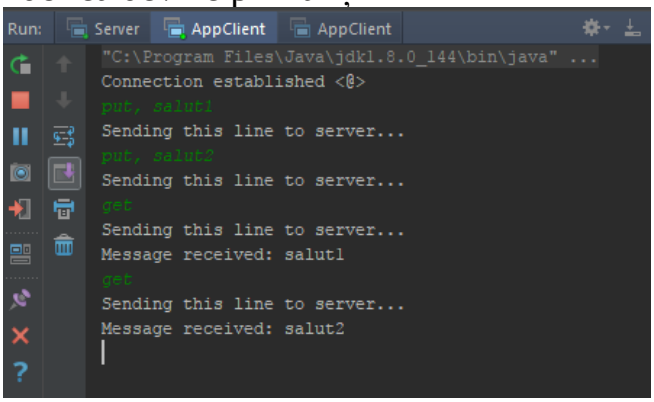


Figura 2 – inserarea mesajului și trimiterea către server

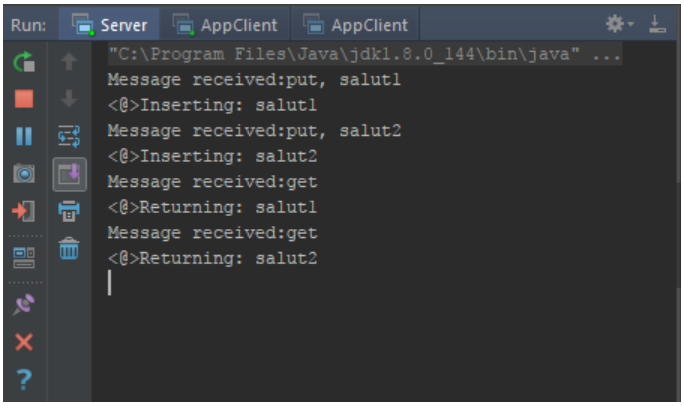


Figura 3 – starea serverului

Concluzie

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 şi 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 corecţie al erorilor, deşi nu este uşor de utilizat şi nici cel mai rapid protocol. Printre avantajele utilizării acestui protocol se numără: este un protocol de reţea 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 şi transfera date între sisteme diferite. Este un cadru de lucru robust, scalabil între platforme client / server. Reprezintă o metodă de acces la resursele interne.

Link la repositoriu: https://github.com/GorduzDaniel/PAD_Lab1