Programare de rețea în Java.

Java network programming.

Obiective:

- înțelegerea conceptelor de programare distribuită;
- implementarea de aplicații client-server TCP și UDP;

Objectives:

- understanding the distributed programming concepts;
- implementing TCP and UDP client-server applications;

Modelul client-server şi programarea în reţea folosind Java

Programarea în reţea folosind limbajul Java este bazată în special pe mecanismul *client-server*. Simplificat acest lucru înseamnă că clientul cere un anumit tip de informaţie, iar serverul îi returnează clientului informaţia cerută.

Conceptul de *socket* în accepţiunea Java

Un *socket* este un canal de comunicaţie care face posibil transferul de date printr-un anumit *port*. Datele se pot transfera prin multiple *socket*-uri dar printr-un singur *port*. Un exemplu în acest sens este faptul că mulţi utilizatori de pe Web accesează *port*ul 80 (HTTP) în acelaşi timp.

Socket-urile în Java sunt divizate în două părţi majore : *DatagramSocket* sau *socket-uri UDP* şi *StreamSocket* sau *socket-uri TCP*.

Socket-uri TCP (stream sockets)

*Socket*urile au fost create ca şi un instrument pentru a realiza mai uşor programarea în reţea. Original din sistemul de operare UNIX, conceptul de socket a fost încorporat în multe medii de programare, inclusiv Java.

Protocolul TCP permite o transmisie sigură a datelor. Şi aici (ca şi la UDP) un *socket* TCP este alocat unei *adrese IP* şi unui număr de *port*. Datele care sunt transmise printr-un *socket* TCP sunt divizate în pachete, fiecare pachet este numerotat şi protejat la erori prin coduri de protecţie. La recepţie aceste pachete sunt verificate din punct de vedere al erorilor şi aliniate în ordine.

Funcţionalitatea acestor *stream socket*uri sunt oferite de clasele *java.net.ServerSocket* şi *java.net.Socket*. Clasa *Socket* facilitează implementarea clientului, în timp ce clasă *ServerSocket* implementează funcţionalitatea unui server.

Un *StreamSocket* seamănă cu o reţea în care legătura este tot timpul activă. Un *StreamSocket* este un *socket* în care datele sunt transferate în mod continuu, prin continuu înţelegându-se că datele nu se transmit neapărat tot timpul, ci *socket*ul în sine este activ pentru comunicaţia respectivă.

Pentru a deschide un *socket,* în Java*,* trebuie specificată o *adresă IP* şi un număr de *port*. Adresele IP sunt reprezentate de clasa *java.net.InetAddress*. Pentru a obţine o adresă *InetAddress* putem folosi metoda statică *getByName*():

Exemplu:

```
InetAddress utcluj = InetAddress.getByName("www.utcluj.ro");
```

sau

```
InetAddress helios = InetAddress.getByName("193.226.5.228");
```

Pentru a afla adresa calculatorului pe care lucrează utilizatorul, se poate utiliza metoda statică *InetAddress.getLocalHost()*. Dar adresa *InetAddress* singură nu creează un *socket,* de aceea trebuie definit şi un număr de *port*. Într-o aplicaţie client-server numai serverul trebuie să ţină seama de *port*ul pe care-l utilizează, pentru că numai acesta trebuie să fie găsit la o locaţie precisă. Deci, se poate specifica numai un număr de *port* sau se poate utiliza *port*ul cu valoarea zero, ceea ce înseamnă utilizarea oricărui port disponibil.

Clasa *Socket*

Patru tipuri de informaţii sunt necesare pentru a creea o comunicare TCP : adresa IP a sistemului local, numărul portului TCP pe care această aplicaţie o utilizează, adresa IP a sistemului cu care se doreşte comunicarea şi numărul portului la care această aplicaţie va răspunde. Clasa Socket este utilizată pentru comunicări în ambele sensuri.

Constructori principali:

- *public Socket (String host, int port) throws UnknownHostException, IOException*
- *public Socket (InetAddress address , int port) throws IOException*
- *public Socket (String host , int port , boolean stream) throws IOException*
- *public Socket (InetAddress address, int port, boolean stream) throws IOException*

Primul constructor permite crearea unui socket prin specificarea numelui destinatarului respectiv portul destinatarului. Al doilea constructor permite crearea unui socket cu obiectul *InetAddress*, obiect care conţine adresa IP a serverului (destinatarului), în timp ce al treilea şi al patrulea constructor sunt similare primelor două, exceptând faptul că ele permit introducerea unei valori booleene, care ne indică protocolul utilizat pentru implementarea socket-ului. Prin definiţie se utilizează TCP, iar dacă valorii booleene îi corespunde valoarea *false* atunci se va utiliza un protocol nesigur cum este UDP.

Obiectul *InetAddress* se poate creea prin mai multe metode statice.

Exemplu:

```
try {
        InetAddress remoteIP = InetAddress.getByName("www.utcluj.ro");
        InetAddress[] allRemoteIPs = InetAddress.getAllByName("www.utcluj.ro");
        InetAddress myIP = InetAddress.getLocalHost();
}
 catch(UnknownHostException excpt) {
        System.err.println("Unknown host: " + excpt);
}
```

Prima metodă returnează adresa IP a www.utcluj.ro . A doua metodă returnează toate adresele IP a www.utcluj.ro, ştiind că unui nume de domeniu îi pot corespunde mai multe adrese IP. Ultima metodă returnează adresa locală a maşinii. Tuturor metodelor prezentate mai sus îi corespunde o excepţie care este tratată în mod corespunzător.

Clasa *Socket* are metode ce permit scrierea şi citirea printr-un *socket*, metodele *getInputStream()* şi *getOutputStream()*.

Pentru a crea mai uşor aplicaţiile, stream-urile returnate de aceste metode sunt ”decorate” cu alte obiecte stream, ca *DataInputStream, BufferedReader, ObjectInputStream,* respectiv *DataOutputStream*, *PrintStream, BufferedWriter, ObjectOutputStream, etc*. Ambele metode *getInputStream()* şi *getOutputStream()* returnează o excepţii care trebuie monitorizate.

Exemplu:

```
try {
        Socket socket = new Socket(" www.utcluj.ro ", 1200);
        DataInputStream input = new DataInputStream(socket.getInputStream());
        PrintStream output = new PrintStream(socket.getOutputStream());
}
```

```
catch(UnknownHostException excpt) {
        System.err.println("Unknown host: " + excpt);
}
catch(IOException excpt) {
        System.err.println("Failed I/O: " + excpt);
}
```

Pentru a scrie un mesaj respectiv pentru a citi unul trebuie utilizate numai streamurile ”decorative”.

Exemplu:

```
output.println("test");
String testResponse = input.readLine();
```

Deci metodele *getInputStream()* şi *getOutputStream()* sunt utilizate pentru a deschide niste canale de comunicație care vor permite citirea sau scrierea informaţiilor de intrare şi ieşire.

Ideea de *stream* are la bază crearea unui canal de comunicaţie între două entităţi, fie ele software sau hardware. Condiţia este ca una dintre entităţi să fie sursă şi cealaltă destinaţie. Astfel sursa va trimite (scrie) informaţia în canalul de comunicaţie (stream), iar destinaţia va prelua (citi) informaţia din canal.

După ce s-a terminat comunicarea prin *socket*, trebuiesc închise instanţele *InputStream* şi *OutputStream*, după care se închide şi socket-ul cu metoda *close()*.

Exemplu:

```
output.close();
input.close();
socket.close();
```

Clasa *ServerSocket*

Pentru a crea un server TCP este necesar să ne referim la o nouă clasă, clasa *ServerSocket*. Această clasă tratează celălalt capăt al comunicaţiei client-server.

ServerSocket permite ataşarea unui port şi aşteptarea conexiunilor de la maşinile client.

Constructori principali:

```
public  ServerSocket (int port) throws  IOException
public  ServerSocket (int port , int count) throws  IOException
```

Primul constructor crează un socket care aşteaptă la portul specificat, permiţând unui număr de 50 de clienţi să aştepte în coada de aşteptare. Al doilea constructor permite schimbarea lungimii cozii de aşteptare, permiţând astfel schimbarea numărului de clienți care aşteaptă pentru a fi procesaţi de server. Variabila *count* specifică o perioadă de timp după care serverul închide conexiunea.

După crearea unui *ServerSocket* se poate utiliza metoda *accept()* pentru a aştepta ca un client să se conecteze, metodă care blochează rularea programului până când se realizează o conexiune. Mai jos este descrisă modalitatea de a creea un *ServerSocket* la portul 2222, acceptarea unei conexiuni şi deschiderea unui *stream* prin care are loc comunicarea îndată ce s-a conectat un client.

Exemplu:

```
try {
        ServerSocket server = new ServerSocket(2222);
        Socket clientConn = server.accept();
        DataInputStream input = new DataInputStream(clientConn.getInputStream());
        PrintStream output = new PrintStream(clientConn.getOutputStream());
}
 catch(IOException excpt) {
        System.err.println("Failed I/O: " + excpt);
}
```

După ce comunicaţia cu clientul a luat sfârşit, serverul trebuie să închidă fluxurile de date, ca mai apoi să închidă şi socket-ul deschis anterior.

Socket-uri UDP

Ca şi la IP nu este garantată livrarea la destinaţie, detecţia erorilor sau secvenţialitatea datagramelor. Avantajul constă în faptul că *DatagramSocket* necesită un număr relativ mic de resurse. Mesajele sunt trimise în pachete individuale şi compacte care pot să nu ajungă la destinaţie într-o anumită ordine sau un anumit timp.

UDP este o alegere bună pentru aplicaţii în care comunicaţia propriu-zisă se poate separa în mesaje discrete. Datele dependente de timp sunt transmise cu aceste protocoale, dar nesiguranţa transmiterii este responsabilitatea programatorului. Cum am menţionat anterior, datele sunt separate în mici mesaje, care încap într-un pachet de o anumită lungime numite şi *datagrame*.

Spre deosebire de protocolul TCP (bazat pe stabilirea unui canal virtual de comunicaţie între aplicaţii), protocolul UDP operează într-un mod orientat pe transmiterea datagramelor. El nu încearcă să stabilească o legătură între aplicaţii, ci doar încapsulează datele într-un pachet UDP pe care îl pasează nivelului IP. La nivelul IP pachetul UDP este ambalat într-o datagramă IP şi transmis nivelului inferior pentru expediţie.

Protocolul UDP nu asigură asamblarea secvenţelor mesajului în ordinea corectă.

Protocolul UDP nu va sesiza pierderea datagramelor pe traseu deoarece nu prevede confirmarea de primire din partea destinatarului. UDP nu retransmite mesajul, misiunea să limitându-se doar la multiplexarea porturilor: preia datele de la porturi, le încapsulează în pachete UDP şi le plasează în canalul de comunicaţie comun IP, respectiv extrage datele din pachetele UDP recepţionate prin acest canal şi le dirijează spre porturile de destinaţie specificate în antetul pachetului.

Protocolul UDP este mai rapid decât protocolul TCP, pentru că elimină timpul pierdut pentru crearea şi distrugerea unei conexiuni. Protocolul UDP îmbunătăţeşte viteza de transfer mai ales în cazul aplicaţiilor care difuzează acelaşi mesaj simultan la mai multe calculatoare dintr-o reţea. Pentru aplicaţii în timp real, cum ar fi aplicaţiile audio/video, garantarea unei transmisii sigure nu este un avantaj, de aceea la acestea se utilizează un protocol neorientat pe conexiuni pentru a minimiza întârzierile.

Aceste caracteristici fac ca protocolul UDP să fie mai simplu şi mai eficient, mai rapid decât protocolul TCP cu singurul dezavantaj că este mai puţin sigur.

Programarea UDP se realizează prin succesiunea următoarelor etape:

- se creează o datagramă adresată care va fi transmisă
- se creează un socket prin care se transmite sau recepţionează datagrame de la o anumită aplicaţie
- inserarea datagramelor în socket pentru a fi transmise
- aşteptarea pentru recepţia datagramelor de la un socket
- decodarea datagramelor pentru a extrage mesajul

Limbajul Java implementează programarea prin SocketDatagram cu ajutorul claselor *DatagramSocket*, *DatagramPacket*, *DatagramChannel*, *MulticastSocket*.

Clasa *DatagramPacket*

Pachetul *java.net* conţine instrumentarul necesar realizării unei comunicaţii UDP. Pentru crearea de datagrame, limbajul java pune la dispoziţia programatorului clasa *DatagramPacket*.

Pentru a crea o datagramă ce va fi trimisă, se foloseşte următorul constructor:

DatagramPacket (byte[] ibuf , int length , InetAddress iaddr , int iport)

unde *ibuf* este tabloul de biţi al mesajului ce urmează a fi trimis, în timp ce *length* este lungimea acestui tablou care determină totodată lungimea datagramei, iar *iaddr* este un obiect *InetAddress* explicat anterior care stochează adresa IP, care este totodată adresa de destinaţie, iar variabilă *port* identifică numărul portului destinatarului la care trebuie să fie trimisă datagrama.

Pentru a recepţiona datagrama, trebuie utilizat un alt constructor în care se stochează datele recepţionate :

public DatagramPacket (byte[] ibuff , int ilength)

unde *ibuff* este un tablou de biţi unde părţile din datagramă vor fi copiate, iar *ilength* conţine numărul de biţi corespunzător lungimii datagramei.

Datagramele nu sunt limitate la o anumită lungime. Utilizatorul poate crea datagrame foarte scurte sau foarte lungi, dar trebuie să se ajungă la o înţelegere între expeditor şi destinatar în ceea ce priveşte lungimea mesajelor transmise pentru că trebuiesc create tablouri de biţi de lungimi apropiate pentru trimiterea, respectiv recepţionarea datagramei. După ce datagrama a fost recepţionată se poate trece la citirea datelor. Metodele referitoare la aceste probleme menţionate anterior, dar şi la alte metode sunt prezentate mai jos.

- *public int getLength()*
- *public byte[] getData()*
- *public InetAddress getAddress()*
- *public int getPort()*

Metoda *getLength()* este utilizată pentru a obţine numărul de bytes conţinuţi în datagramă. Metoda *getData()* este utilizată pentru a obţine un tablou de bytes conţinând datele recepţionate, iar metoda *getAddress()* este utilizată pentru a obţine un obiect *InetAddress* care identifică expeditorul, în timp ce *getPort()* indică portul UDP utilizat.

Clasa *DatagramSocket*

Transmiterea şi recepţionarea acestor datagrame se realizează cu clasa *DatagramSocket*, care creează un socket UDP. Sunt utilizaţi doi constructori, unul permiţând sistemului să aloce (atribuie) un port dinamic, un port neutilizat, în timp ce celălalt permite utilizatorului să specifice un port cunoscut:

public DatagramSocket () throws SocketException;
public DatagramSocket (int port) throws SocketException;

Primul constructor este constructorul de bază care nu are nici un parametru, iar al doilea constructor creează un *DatagramSocket* conectat la un port specificat.

După ce *socke*tul a fost creat se poate transmite sau recepţiona datagrame, utilizând metodele *send()* şi *receive()* care au următoarele prototipuri:

- *public void send (DatagramPacket p) throws IOException*
- *public synchronized void receive (DatagramPacket p) throws IOException*

Pachetele în sine sunt reprezentate de obiecte *java.net.DatagramPacket*, pe care le putem construi printr-un tablou de bytes. Metoda *send()* are nevoie ca *DatagramPacket* să fie corect construit şi să fie încărcat corect cu: datele care urmează a fi trimise, lungimea şirului de date, adresa *InetAddress* a destinatarului şi *port*ul destinatarului, în timp ce metodă *receive()* are nevoie ca numai un buffer de date şi lungimea datelor să fie definite. Lungimea datelor recepţionate sunt accesibile prin metoda *getLength().*

Lucru individual

1. Să se scrie o aplicație client-server în cadrul căreia clientul trimite două mesaje ce conțin numele vostru și grupa din care faceți parte. Serverul va răspunde cu mesajul: ”Salut [nume] din grupa [grupa]". Folosiți o conexiune cu confirmare (TCP/IP) și una fără confirmare (UDP).

==========

2. Să se scrie o aplicație Java distribuită în cadrul căreia serverul așteaptă conexiuni fără confirmare prin care clienții verifică disponibilitatea unor fișiere pe sistemul local al serverului. Serverul va răspunde cu mesajul: "Fișierul [nume_fișier] este/nu este disponibil pe server"

==========

3. Creați o aplicație Java de tip *Frame* ce conține un câmp de tip *TextArea* și unul de tip *Label*. Aplicația va prelua textul din cadrul câmpului de text și-l va trimite unui server. Serverul va răspunde cu un mesaj ce specifică numărul de caractere alfabetice și numerice conținute în mesajul primit. Răspunsul serverului va fi afișat în câmpul etichetă. Puteți folosi oricare din tehnlogiile AWT sau Swing.

==========

4. Să se realizeze o aplicație de tip client-server în care clientul poate solicita o serie de informații de la server folosind o serie de comenzi predefinite, de exemplu: /getProperties - returnează proprietețile sistemului server (utilizați *System.getProperties().toString()*), /getTime - returnează ora locală a serverului (utilizați *Date().toString()*), /getConnection - returnează adresa clientului. Conexiunea cu serverul este oprită prin comanda #exit.

==========

5. Creați o aplicație Java de tip client-server ce permite transferul unui fișier binar de la server către client. Clientul va solicita un anumit fișier, iar serverul va trimite conținutul acestuia printr-o conexiune TCP/IP. Clientul va recompune fișierul pe sistemul local și-l va salva tot în format binar.

==========

6. Creați o aplicație Java de tip group chat între 2 sau mai mulți clienți conectați la un server. Serverul va aștepta conectare clienților identificați printr-un id unic format din 6 cifre aleatoare. Clienții vor trimite mesaje către server, iar acesta le va distribui celorlați clienți conectați și va include în fața mesajului id-ul clientului care l-a trimis.

==========

7. Să se realizeze o aplicație client-server prin intermediul căreia clientul poate solicita numerele matricole ale studenților din UTC-N. Informațiile sunt stocate într-un fișier text pe server și au formatul:

Nume, Prenume, Specializarea, An, Grupa, Nr. matricol

Clientul va trimite interogări către server de forma:

?nume="Ionescu"&prenume="Ion"&specializarea="TST"&an=3&grupa=2031

Mesajele clientului nu trebuie să conțină toate câmpurile (pot să fie compuse doar din nume și prenume sau doar grupa, etc.). Serverul va trimite înapoi în format text toate numerele matricole corespunzătoare interogării primite de la client.

Individual work

1. Write a client-server Java application in which the client sends two messages which contain your name and group. The server will respond with the message: "Hi [name] from group [group]!". Use both a TCP/IP and a UDP protocol connection.

==========

2. Write a distributed Java application in which the server waits for connections without confirmation through which the clients check if some specific files exist on the server's local system. The server will respond with the message "The file [filename] is/is not available on the server".

==========

3. Write a *Frame* application which contains a *TextArea* and a *Label* field. The application will take the text from the *TextArea* and send it to the server. The server will respond with a message which specifies the number of alphabetic and numeric characters contained in the received string message. The response of the server will be displayed in the *Label* field of the applet. You can use any of the AWT or Swing approaches.

==========

4. Write a Java client-server application in which the client can request several specific information from the server by using a set of predefined commands. For example: /getProperties - returns the properties of the server's system (use *System.getProperties().toString()*), /getTime - returns the local time of the server (use *Date().toString()*), /getConnection - returns the address of the client. The connection with the server is stopped by sending the #exit command.

==========

5. Write a client-server Java application which enables the server to send a binary file to the client. The client will send a request for a specific file, and the server will send its content over a TCP/IP connection. The client will recompose the file and save it on the local file system.

==========

6. Write a Java program which implements a group chat application between 2 or more clients connected to a server. The server will wait for connections from clients identified by a unique 6 character id composed only of digits. The clients will send messages to the server, which will be in turn distributed to all the connected clients. The messages will also contain the id of the client who sent the message.

==========

7. Build a client- server application through which a client can request the registration numbers of the students from TUC-N. The information is stored in a text file on the server and have the following format:

Surname, Name, Track, Year, Group, RegistrationNumber

The client will send queries to the server in the following format:

?surname="Ionescu"&name="Ion"&track="TST"&year=3&group=2031

The client's messages do not have to contain all the fields (I.e. can be composed only of the surname, or track, or group, etc.) The server will send back all the registration numbers in text format, corresponding to the query received from the client.