**Socket-uri**

**Modelul Client/Server**

După cum ştim, comunicarea între nodurile unei reţele constă în transmiterea (recepţionarea) de pachete către (de la) gazde ale aceleaşi reţele sau ale unei alte reţele.

Modelul utilizat pe scară largă în sistemele distribuite (şi care va fi cel folosit în continuare) este *sistemul Client/Server*. În acest model există:

- o mulţime de *procese Server*, fiecare jucând rolul de gestionar de resurse pentru o colecţie de resurse de un anumit tip;

- o mulţime de *procese Client*, fiecare executând activităţi care necesită acces la resurse hard/soft disponibile (partajate) de servere.

Un gestionar de resurse poate avea şi el nevoie de resurse gestionate de un alt proces *Server*. Drept urmare, unele procese pot fi atât de tip *Client*, cât şi de tip *Server*. Dar doar serverele gestionează resurse.

Serverele sunt cele care îşi încep primele activitatea. În mod tipic, un server oferă succesiv clienţilor posibilitatea de a se conecta la el (spunem că *acceptă conexiuni* de la clienţi). La început clientul îşi manifestă dorinţa de a se conecta şi, dacă serverul este gata să accepte conexiunea, aceasta se realizează efectiv; este vorba deci de o acţiune de la client către server. Apoi transmisia informaţiilor devine bidirecţională, fluxul de informaţii putând circula acum în ambele sensuri.

Teoretic, activitatea unui server se desfăşoară la infinit.

Internetul foloseşte *nume (adrese) simbolice* pentru reţele şi pentru maşinile gazdă; ele se mai numesc *nume de domeniu*. Lor li se asociază în mod biunivoc *adrese (numerice) IP*, folosite efectiv la comunicarea între nodurile reţelei. Asocierea cade în sarcina unui *sistem de nume de domeniu* (*DNS* = Domain Name System).

O adresă numerică IP are foma:

a.b.c.d

unde a, b, c, d sunt numere naturale din mulţimea {0, 1, ..., 255}.

Pentru conectare la server, clientul trebuie să cunoască adresa serverului (fie cea numerică, fie cea simbolică), precum şi numărul *portului* pus la dispoziţie de server. Portul nu este o locaţie fizică, ci o extensie software corespunzătoare unui serviciu. Serverul poate oferi mai multe servicii, pentru fiecare fiind alocat un (număr de) port. Dintre porturile standard amintim:

7 : *ping* (ecou) 21 : *ftp* (transfer de fişiere) 23 : *telnet*

25 : *email* 79 : *finger* 80 : *Web*

110 : *POP3* (Post Office Protocol)

Porturile din intervalul 0..1023 sunt în general rezervate pentru servicii speciale (de genul celor de mai sus) şi pentru clienţi privilegiaţi.

Clientul iniţiază conexiunea prin reţea, specificând adresa serverului şi portul prin care doreşte să comunice. Serverul trebuie să precizeze numai portul; apoi el trebuie să execute o comandă prin care să anunţe că este gata să accepte conexiuni pe portul respectiv; drept urmare el rămâne în aşteptare până când un client doreşte să se conecteze şi conexiunea este stabilită cu succes. Un server poate accepta conexiuni de la mai mulţi clienţi: pentru fiecare crează un fir de executare.

**Clasa InetAddress**

Această clasă, aflată în pachetul java.net, furnizează informaţii asupra adreselor (simbolică şi numerică) unui calculator gazdă. Amintim numai metoda:

public static InetAddress getLocalHost()

care returnează numele calculatorului gazdă (pe care se află în curs de executare aplicaţia). Acesta, convertit la String, are forma: *adresă\_simbolică*/*adresă\_IP*.

***Exemplul 1*.** Determinarea adreselor calculatorului gazdă se face simplu astfel:

import java.net.\*;

class Adrese {

public static void main(String[] args) {

try {

System.out.println(InetAddress.getLocalHost());

}

catch(UnknownHostException e) {

System.out.println("Gazda nu are adresa IP");

}

}

}

*Întrebare*. "*Are sens să studiem facilităţile oferite de Java pentru lucru în reţea dacă avem la dispoziţie un singur calculator*?". DA! Putem de exemplu deschide mai multe ferestre de comandă, fiecare corespunzând unui calculator din reţea. Drept adresă IP putem folosi numele calculatorului, numele standard "localhost" sau "127.0.0.1". Este chiar indicat să verificăm în astfel programele pe care le scriem. Desigur, este vorba doar de o primă verificare, pentru că lucrul în reţea implică şi alte aspecte, care nu pot fi surprinse în acest mod de lucru.

**Comunicare prin socket-uri**

**Socket-uri**

Socket-urile sunt obiecte folosite pentru transmiterea de date folosind protocolul TCP/IP. Ele sunt obiecte ce trebuie create la ambele capete ale conexiunii. Socket-urile client sunt obiecte de tipul clasei Socket, iar socket-urile server sunt obiecte de tipul clasei ServerSocket; ambele clase fac parte din pachetul java.net. Socket-urilor li se pot ataşa un flux de intrare şi unul de ieşire, prin care pot recepţiona/ transmite, date.

Prezentăm în continuare schema generală a lucrului cu socket-uri.

Un mod tipic de creare a unui *socket client* este următorul:

Socket cs = null;

InputStream is = null; OutputStream os = null;

try {

cs = new Socket(*adresa*,*nrport*);

is = cs.getInputStream(); os = cs.getOutputStream();

}

catch(UnknownHostException e) { ... }

catch(IOException e) { ... }

unde adresa este adresa IP a serverului, iar nrport este numărul portului ales pentru comunicare. Socket-ului îi sunt ataşate fluxul os ce va fi folosit pentru a transmite date serverului, precum şi fluxul is ce va fi folosit pentru recepţionarea datelor transmise de server. După caz, fluxurile vor fi înzestrate cu facilităţile de transmitere a datelor primitive, a obiectelor etc.

Un mod tipic de creare a unui *socket server* este următorul:

ServerSocket ss = null; Socket cs = null;

try {

ss = new ServerSocket(nrport);

// System.out.println("Serverul a pornit");

}

catch(IOException e) { ... }

InputStream is = null; OutputStream os = null;

try {

cs = ss.accept();

is = cs.getInputStream(); os = cs.getOutputStream();

}

catch(UnknownHostException e) { ... }

catch(IOException e) { ... }

Observăm că pentru server nu este necesară precizarea unei adrese IP, ci numai a unui port, care trebuie să coincidă cu cel folosit de client.

Metoda accept lucrează astfel: se aşteptă ca un client să încerce să se lege la server pe portul precizat; în momentul în care acest lucru se întâmplă şi legătura s-a stabilit, metoda creează şi întoarce un socket cs de tip client. Acestuia îi ataşăm un flux de intrare şi unul de ieşire. Ca şi pentru client, fluxurile pot fi înzestrate cu facilităţile de transmitere a datelor primitive, a obiectelor etc.

cs

os

is

cs

ss

is

os

##### Client

##### Server

În final toate socket-urile şi fluxurile trebuie închise explicit prin invocarea metodei close.

**Clasele Socket şi ServerSocket**

Clasa Socket din pachetul java.net:

public class Socket extends Object

oferă constructori publici şi metode publice pentru lucrul cu socket-urile client. Dintre ele menţionăm următoarele:

Socket(String gazda, int port)

throws IOException,UnknownHostException

creează un socket şi îl conectează la portul port al serverului cu numele simbolic gazda;

Socket(InetAddress adresa, int port) throws IOException

creează un socket şi îl conectează la portul port al serverului cu adresa adresa;

InputStream getInputStream() throws IOException

întoarce un flux de intrare pentru socket-ul curent;

OutputStream getOutputStream() throws IOException

întoarce un flux de ieşire pentru socket-ul curent;

void close() throws IOException

închide (distruge) socket-ul curent.

Clasa ServerSocket din pachetul java.net:

public class ServerSocket extends Object

oferă un constructor public, precum şi metode publice pentru lucrul cu socket-urile server. Dintre ele menţionăm următoarele:

ServerSocket(int port) throws IOException

creează un socket de tip server la portul port;

Socket accept() throws IOException

aşteaptă realizarea unei conexiuni la socket-ul server curent. În momentul în care conexiunea este realizată, metoda creează şi întoarce un socket de tip client;

void close() throws IOException

închide (distruge) socket-ul curent.

***Exemplul 2****. Dorim să aflăm dacă un server având o adresă dată oferă sau nu servicii pe un anumit port.*

Programul se reduce la încercarea de a crea un socket de tip client pentru adresa şi portul citite de la intrarea standard.

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

public class WebCheck {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Adresa: "); String adresa = sc.next();

System.out.print("Portul: "); int port = sc.nextInt();

String out = "Serverul " + adresa;

try {

Socket web = new Socket(adresa,port);

out += " ofera servicii pe portul " + port;

web.close();

}

catch(IOException e) {

out += " nu ofera servicii pe portul " + port;

}

System.out.println(out);

}

}

De exemplu, dacă pornim serverul Tomcat pe portul 8080 şi specificăm drept adresă localhost, iar ca port 8080, va fi afişat mesajul:

Serverul localhost ofera servicii pe portul 8080

Prezentăm în continuare un server rudimentar care oferă servicii pe un port citit de la intrare; serviciile se reduc la semnalarea conectării clienţilor:

import java.net.\*; import java.util.\*;

import java.io.\*;

class ServerRud {

public static void main(String[] abc) throws IOException {

Scanner sc = new Scanner(System.in);

System.out.println("Portul : ");

ServerSocket ss = new ServerSocket( sc.nextInt() );

while (true) {

ss.accept();

System.out.println("S-a conectat un nou client");

}

}

}

***Exemplul 3*.** *Chat (conversaţie) între două calculatoare.*

Pentru simplificare, considerăm că fiecare mesaj trimis de la un calculator la celălalt se reduce la o linie de text.

Clientul este cel care începe discuţia, transmiţând un mesaj şi aşteptând să primească mesajul de răspuns, activităţi care se repetă până când el transmite mesajul "STOP". Serverul repetă şi el succesiunea de activităţi recepţie + transmisie până când primeşte mesajul "STOP".

Clientul va folosi unitatea de compilare Chat\_C.java, iar serverul va folosi unitatea de compilare Chat\_S.java.

// Unitatea de compilare Chat\_C.java

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

class Chat\_C {

public static void main(String[] sir) throws IOException {

Scanner sc = new Scanner(System.in);

System.out.print("Adresa serverului si portul :");

String adresa = sc.next(); int port = sc.nextInt();

sc.nextLine();

Socket cs = null;

try { cs = new Socket(adresa,port); }

catch(Exception e) {

System.out.println("Conexiune esuata");

System.exit(1);

}

DataInputStream dis; DataOutputStream dos;

dis = new DataInputStream(cs.getInputStream());

dos = new DataOutputStream(cs.getOutputStream());

String linie;

for( ; ; ) {

System.out.print("Mesaj de trimis : \t");

linie = sc.nextLine(); dos.writeUTF(linie);

if( linie.equals("STOP") ) break;

linie = dis.readUTF();

System.out.println("Mesaj receptionat :\t" + linie);

}

System.out.println("GATA");

cs.close(); dis.close(); dos.close();

}

}

// Unitatea de compilare Chat\_S.java

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

class Chat\_S {

public static void main(String[] sir) throws IOException {

ServerSocket ss = null; Socket cs = null;

Scanner sc = new Scanner(System.in);

System.out.print("Portul : ");

ss = new ServerSocket(sc.nextInt());

sc.nextLine();

System.out.println("Serverul a pornit !");

cs = ss.accept();

DataInputStream dis; DataOutputStream dos;

dis = new DataInputStream(cs.getInputStream());

dos = new DataOutputStream(cs.getOutputStream());

String linie;

for( ; ; ) {

linie = dis.readUTF();

System.out.println("Mesaj receptionat :\t" + linie);

if( linie.equals("STOP") ) break;

System.out.print("Mesaj de trimis :\t");

linie = sc.nextLine();

dos.writeUTF(linie);

}

cs.close(); dis.close(); dos.close();

}

}