Programarea în reţea – 1

1. Scopul lucrării

Scopul acestei lucrări este insuşirea tehnicilor de programare în reţea în limbajul Java utilizând socket-uri.

2. Noţiuni preliminare

Calculatoarele conectate în reţea comunică între ele utilizând protocoalele TCP (Transport Control Protocol) şi UDP (User Datagram Protocol) conform diagramei:

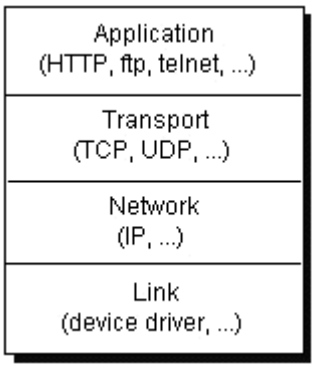


Figura 1. Nivelele de comunicare în reţea

Pentru realizarea unor programe care comunică în reţea în java, se utilizează clasele din pachetul ***java.net*** . Acest pachet oferă clasele necesare pentru realizarea unor programe de reţea independente de sistemul de operare.

In tabelul următor sunt prezentate principalele clase care sunt utilizate pentru construirea unor programe de reţea.

Class	*Scop*
URL	Reprezintă un URL
URLConnection	Returnează conţinutul adresat de obiectele URL
Socket	Crează un socket TCP
ServerSocket	Crează un socket server TCP
DatagramSocket	Crează un socket UDP
DatagramPacket	Reprezintă o datagrama trimisă printr-un obiectDatagramSocket
InetAddress	Reprezintă numele unui pc din reţea, respectiv IP-ul corespunzător

Java oferă două abordări diferite pentru realizarea de programe de reţea. Cele două abordări sunt asociate cu clasele:

- Socket, DatagramSocket şi ServerSocket
- URL, URLEncoder şi URLConnection

Programarea prin socket-uri reprezintă o abordare de nivel jos, prin care, două calculatoare pot fi conectate pentru a realiza schimb de date. Ca principiu de bază, programarea prin socketuri face posibilă comunicarea în mod full-duplex între client şi server. Comunicarea se face prin **fluxuri de octeţi**.

Pentru ca să se desfăşoare corespunzător comunicarea, programatorul va trebui să implementeze un protocol de comunicaţie (reguli de dialog), pe care clientul şi serverul îl vor urma.

Identificarea unui calculator în reţea

Orice calculator conectat la Internet este identificat in mod unic de adresa sa **IP** (IP este acronimul de la **Internet Protocol**). Adresele IP sunt identificatori unici ai dispozitivelor aflate în rețea, folosiți pentru a stabili canale de comunicatie.

1. O adresă IP poate fi **statică** sau **dinamică**.

O adresă IP statică este una pe care trebuie să o configurezi manual prin intermediul setărilor de rețea.

O adresă dinamică este alocată de către Dynamic Host Configuration Protocol (DHCP), de obicei pentru o perioadă de timp limitată.

2. O adresa IP poate fi **numerica** sau **simbolica**.

Pentru **adresele numerice** in prezent, există două standarde relevante :

IP versiunea 4 (**IPv4**) și IP versiunea 6 (**IPv6**).

IPv4 reprezintă un număr reprezentat 8 octeţi, cum ar fi de exemplu:193.226.5.33.

Adresele IPv4 sunt împărțite în trei categorii (A,B,C), numite clase. Diferența principală dintre clase o reprezintă numărul de biți alocați pentru identificarea rețelei și a

stației de lucru. Clasa din care face parte o adresă IPv4 poate fi identificată conform primilor biți din scrierea binară a primului număr din notația zecimală.

Adresele IPv4 din clasa D sunt folosite pentru adresarea *multicast*.

Multicasting se referă la comunicarea în grup.

Adresele IPv4 din *clasa E* nu pot fi folosite fiind sunt rezervate in scopuri experimentale.

Adresele IPv6 sunt compuse din opt grupuri de caractere separate prin două puncte (:). Spre deosebire de adresele IPv4, acestea pot conține și litere de la *a* la *f.*

Adresele IPv6 sunt destul de greu de gestionat, așa că există câteva reguli care simplifică modul de scriere. Dacă unul sau mai multe grupuri sunt "0000", zerourile pot fi omise și înlocuite cu două puncte dublate (::), iar zerourile de la începutul unui grup pot fi, de asemenea, omise. În plus, față de IPv4, adresele IPv6 nu sunt împărțite în clase.

Corespunzătoare unei adrese numerice există şi o adresă IP **simbolică**, cum ar fi **utcluj.ro**. De asemenea fiecare calculator aflat într-o reţea locală are un nume unic ce poate fi folosit la identificarea locală a acestuia.

Clasa Java care reprezintă noţiunea de adresă IP este ***InetAddress***. Pentru a construi un obiect se foloseşte comanda:

```
InetAddress address =InetAddress.getByName("121.3.1.2");
```

Pentru a vedea toate modurile în care pot fi construite obiecte de tip ***InetAddress*** studiaţi documentaţia acestei clase.

Un calculator are în general o singură legătură fizică la reţea. Orice informaţie destinată unei anumite maşini trebuie deci să specifice obligatoriu adresa IP a acelei maşini. Insă pe un calculator pot exista concurent mai multe procese care au stabilite conexiuni în reţea, asteptând diverse informaţii. Prin urmare datele trimise către o destinaţie trebuie să specifice pe lângă adresa IP a calculatorului şi procesul către care se îndreaptă informaţiile respective. Identificarea proceselor se realizează prin intermediul porturilor.

Orice aplicaţie care comunică în reţea este identificată în mod unic printr-un port, astfel încât pachetele sosite pe calculatorul gazdă să poată fi corect rutate către aplicaţia destinaţie.

Valorile pe care le poate lua un număr de port sunt cuprinse între 0 şi 65535 (deoarece sunt numere reprezentate pe 16 biţi). Porturile cuprinse între 0 şi 1023 sunt rezervate unor servicii sistem, deci nu se recomandă folosirea acestora.

Definitia socket-ului: Un socket reprezintă un punct de conexiune într-o reţea TCP\IP. Când două programe aflate pe două calculatoare în reţea doresc să comunice, fiecare dintre ele utilizează un socket. Unul dintre programe (serverul) va deschide un socket şi va aştepta conexiuni, iar celălalt program (clientul), se va conecta la server şi astfel schimbul de informaţii poate începe. Pentru a stabili o conexiune, clientul va

trebui să cunoască adresa destinaţiei (a pc-ului pe care este deschis socket-ul) şi portul pe care socketul este deschis.

Principalele operaţii care sunt făcute de socket-uri sunt:

- acceptare conexiuni
- conectare la un alt socket
- trimitere date
- recepţionare date
- închidere conexiune.

3. Aplicaţie client-server cu server monofir

Pentru realizarea unui program client-server se utilizează clasele ServerSocket şi Socket.

Programul server va trebui să deschidă un port şi să aştepte conexiuni. In acest scop este utilizată clasă ***ServerSocket***. In momentul în care se crează un obiect ***ServerSocket*** se specifică portul pe care se va iniţia aşteptarea. Inceperea ascultării portului se face apelând metoda ***accept()***. In momentul în care un client s-a conectat, metoda ***accept()*** va returna un obiect Socket.

La rândul său clientul pentru a se conecta la un server, va trebui să creeze un obiect de tip Socket, care va primi ca parametri adresa serverului şi portul pe care acesta aşteaptă conexiuni.

Atât la nivelul serverului cât şi la nivelul clientului, odată create obiectele de tip Socket, se vor obţine fluxurile de citire şi de scriere. In acest scop se utilizează metodele ***getInputStream()*** şi ***getOutputStream().***

In listingul următor este prezentat programul server.

```
import java.net.*;
import java.io.*;

public class ServerSimplu {
  public static void main(String[] args) throws IOException{

    ServerSocket ss=null;
    Socket socket=null;
    try{
      String line="";
      ss = new ServerSocket(1900);    //crează obiectul serversocket
      System.out.println("astept conexiuni…");

      socket = ss.accept(); //incepe aşteptarea pe portul 1900

      //în momentul în care un client s-a  conectat ss.accept() returnează
      //un socket care identifică conexiunea
       //crează fluxurile de intrare/ieşire
```

```
      BufferedReader in = new BufferedReader(
           new InputStreamReader(socket.getInputStream()));

      PrintWriter out = new PrintWriter(
           new BufferedWriter(new OutputStreamWriter(
             socket.getOutputStream())),true);

      while(!line.equals("END")){
        line = in.readLine();                //citeste datele de la client
        out.println("ECHO "+line);       //trimite date la client
      }

    }catch(Exception e){e.printStackTrace();}
     finally{
      ss.close();
      if(socket!=null) socket.close();
     }
  }
}
```

Programul client este prezentat în listingul următor:

```
import java.net.*;
import java.io.*;

public class ClientSimplu {

  public static void main(String[] args)throws Exception{
    Socket socket=null;
    try {
    //creare obiect address care identifică adresa serverului

      InetAddress address =InetAddress.getByName("localhost");

    //se putea utiliza varianta alternativă: InetAddress.getByName("127.0.0.1")

    socket = new Socket(address,1900);

      BufferedReader in =  new BufferedReader(new InputStreamReader(
                                                  socket.getInputStream()));
      // Output is automatically flushed  by PrintWriter:

      PrintWriter out =  new PrintWriter( new BufferedWriter(
                                  new OutputStreamWriter(socket.getOutputStream())),true);

      for(int i = 0; i < 10; i ++) {
        out.println("mesaj " + i);
        String str = in.readLine();       //trimite mesaj
        System.out.println(str);         //aşteaptă răspuns
      }
      out.println("END");        //trimite mesaj care determină serverul să închidă conexiunea
```

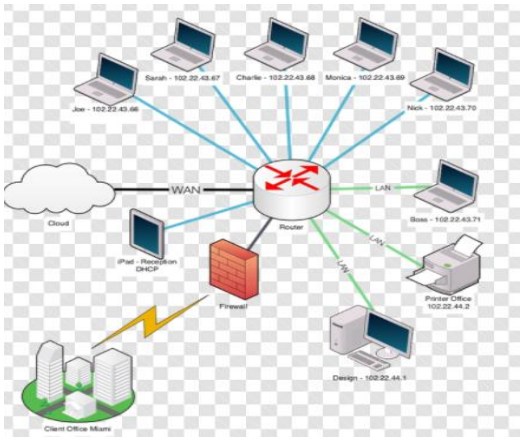
```
    }
    catch (Exception ex) {ex.printStackTrace();}
    finally{
      socket.close();
    }
  }
}
```

Pentru verificare se va starta serverul, după care se va starta clientul.

4. Port Forwarding

4.1. Arhitectura Network si vizibilitatea IP

Arhitectura networks are de cele mai multe ori o subrețea de tip stea (LANs = Local Area Networks), unde in centru se afla de obicei un router (sau server). Acest nod (router) este punctul de conexiune a subrețelei (LAN) cu restul domeniului (internet) si asigura transferul datelor din nodurile exterioare (ex. calculatoare) spre subrețea si invers. Fiecare nod din reţea, inclusiv router-ul pot avea protecţie de tip firewall.



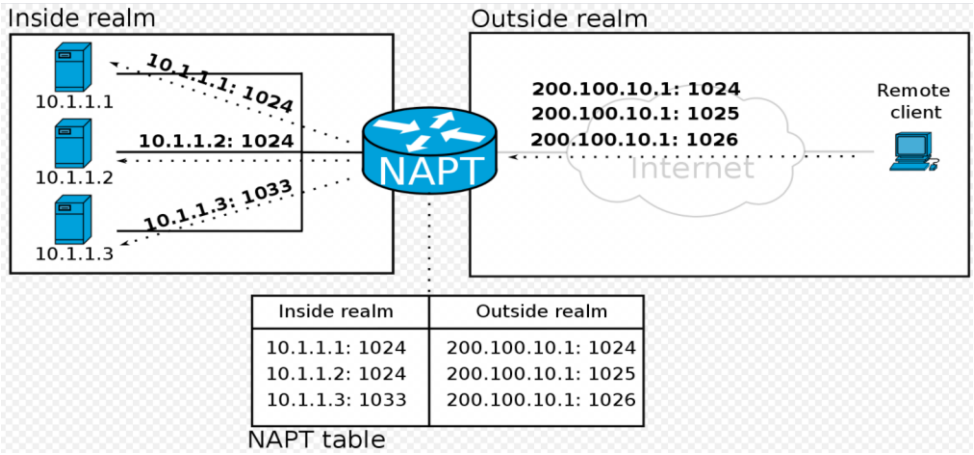
Fiecare nod din LANs este in mod unic identificat prin adresa IP de obicei sub forma numerica: 192.168.0.5, etc, care este alocata dinamic (se poate schimba când un nod intra sau părăsește sistemul). In subretele diferite de tip LANs, același IP poate sa corespunda unor calculatoare diferite. Intr-o subrețea LAN router-ul are o adresa IP fixa, de obicei de tipul: 192.168.0.1. Nodurile din LAN nu sunt vizibile in afara reţelei. .

In exteriorul reţelei LAN, sunt adrese IP statice si dinamice. Router-ul de asemenea are o adresa IP dinamica de forma 187.25.148.5 (Ipv4), care poate fi folosite pentru a identifica router-ul din afara LAN-ului.

Pentru ca nodurile din reţeaua LAN nu sunt vizibile in afara (doar router-ul este vizibil), conexiunea TCP/IP nu se poate realiza, adresa IP nu poate fi identificata.

4.2. Port forwarding

Router-ul poate fi configurat sa redirecționeze cererea (sau orice transfer de date) de la un port către un nod (calculator) din subrețeaua LAN, adică sa proceseze toate cererile. Operația se numește **port forwarding**,(redirecționarea portului) o aplicație din NAT (Network address translation). Prin aplicaţii complexe, porturile pot fi redirecționate către diferite noduri din subrețeaua LAN, (după cum se poate vedea in figura alăturata) rezultând o tabela de routare de următoarea forma NAPT = Network Address Port Translation.



4.3. Configurarea portului forwarding pe router

Găsește **router IP in interiorul LAN**: open **cmd**, type *ipconfig*, select the *Default Gateway*.

Găsește **computer IP**: open **cmd**, type *ipconfig*, select *IPv4 Address*.

Găsește **router IP așa cum se vede in afara LAN**: use **https://whatismyipaddress.com/**

Accesează pagina de configurare a router-ului prin tastarea IP-ului router-ului din subrețeaua LAN in browser.

Configurarea diferă pentru fiecare model de router dar setarile sunt la fel!!

Deschide tabela **Forward Rules -> Port Mapping Configuration**.

Aici creează o regula noua cu *Enable Port Mapping* verifica si scrie urmatoarele detalii:

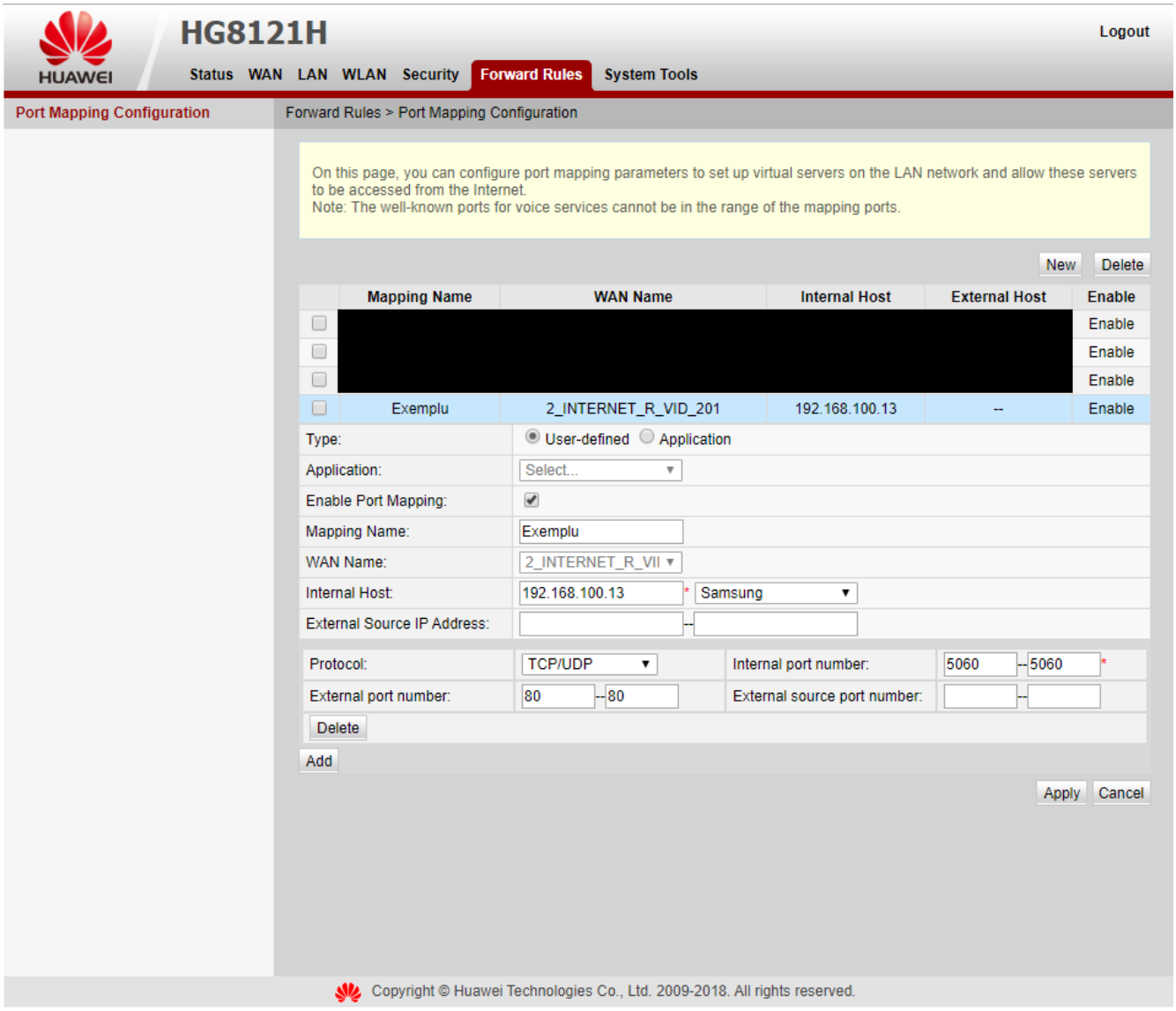
Internal Host = IP calculatorului (se poate selecta host-ul de pe dropdown)

External Source IP Address = IP calculatorului din afara reţelei, care va fi accesat (pentru opțiunea empty => orice IP din afara va avea acces). Deocamdată îl lăsăm așa.

Protocol = UTP/TCP

Internal port number = portul de pe calculator care va fi deschis (5060 de exemplu)

External port number = portul de pe router care va fi folosit din afara retelei (80 de exemplu)



Aplicația TCP va fi configurata in modul următor:

1. serverul rulează pe un calculator din interiorul subrețelei LAN cu IP = **computer** IP. Serverul va începe sa asculte pe un port *Internal port number* (5060 de exemplu)

2. aplicația clientului se va conecta la server folosind **router IP așa cum se vede din exteriorul subrețelei LAN**, folosind portul *External port number* (80 de exemplu)

NOTA1: Unele porturi pot fi blocate de *InternetServiceProvider*. In acest caz, selectează alt port.
NOTA2: Se poate ca protecția firewall a calculatorului (sau a router-ului) sa blocheze comunicarea. In acest caz dezafectati firewalls (sau folositi advanced user, setati TCP application as exception for the firewall).

5. Aplicaţie client-server cu server multifir

Analizând programul server prezentat în secţiunea anterioară se observă că acesta poate servi doar un singur client la un moment dat. Pentru ca serverul să poată servi mai mulţi clienţi simultan, se va utiliza programarea multifir.

Ideea de bază este simplă, şi anume, serverul va aştepta conexiuni prin apelarea metodei ***accept().*** In momentul în care un client s-a conectat şi metoda ***accept()*** a returnat un Socket, se va crea un fir de execuţie care va servi respectivul clientul, iar severul va reveni în aşteptare.

In programul următor este prezentat un server multifir – capabil de a servi mai multi clienţi simultan.

```
import java.io.*;
import java.net.*;

public class ServerMultifir {
        public static final int PORT = 1900;
        void startServer()   {
                ServerSocket ss=null;
                try   {
                ss = new ServerSocket(PORT);
                while (true)  {
                        Socket socket = ss.accept();
                        new TratareClient(socket).start();
                }

                }catch(IOException ex)
                {
                        System.err.println("Eroare :"+ex.getMessage());
                }
                finally  {
                        try{ss.close();}catch(IOException ex2) { }
                }
        }
         public static void main(String args[])   {
                ServerMultifir smf = new ServerMultifir();
                smf.startServer();
```

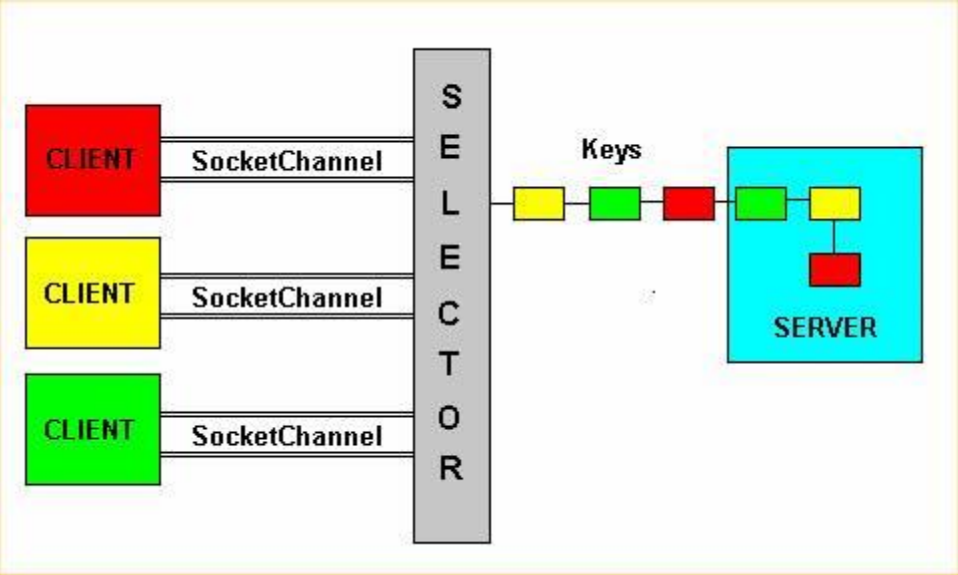
```
            }
}
class TratareClient extends Thread {
            private Socket socket;
            private BufferedReader in;
            private PrintWriter out;
            TratareClient(Socket socket)throws IOException
            {
                        this.socket = socket;
                          in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                          out = new PrintWriter(new BufferedWriter(
                                    new OutputStreamWriter( socket.getOutputStream())));
            }

            public void run() {
            try {
            while (true) {
               String str = in.readLine();
                     if (str.equals("END")) break;
               System.out.println("Echoing: " + str);
               out.println(str);
            }//.while
            System.out.println("closing...");
            }
            catch(IOException e) {System.err.println("IO Exception");}
            finally {
        try {
            socket.close();
            }catch(IOException e) {System.err.println("Socket not closed");}
    }
            }//.run
}
```

6. Socket-uri neblocante

Versiunea Java 2 Standad Edition 1.4 introduce un nou mecanism de comunicare în reţea prin interemediul **socket-urilor neblocante** – acestea permit comunicarea între aplicaţii fără a bloca procesele în apeluri de metode destinate deschiderii unei conexiuni, citirii sau scrierii de date.

Soluţia clasică pentru a construi o aplicaţie server care deserveşte mai mulţi clienţi este de a utiliza tehnologia firelor de execuţie şi de a aloca câte un fir de execuţie pentru fiecare client deservit. Folosind tehnologia socket-urilor neblocante programatorul va putea implementa o aplicaţie server pentru deservirea clienţilor fără a fi nevoit să apeleze în mod explicit la fire de execuţie pentru tratarea cererilor clienţilor. În continuare vor fi prezentate principiile de bază ale acestei tehnologii şi modul în care pot fi construite aplicaţii client şi aplicaţii server pe tehnologia socket-urilor neblocante.

Arhitectura unui sistem ce utilizează socketuri neblocante pentru comunicare este ilustrată în figura următoare.



Arhitectură cu socketu-uri neblocante.

Principalele operaţii ce au loc în cadrul unei aplicaţii bazată pe această tehnologie sunt:

- Clientul: trimite cereri către server
- Serverul: recepţionează cereri
- SocketChannel: permite transmiterea de date între client şi server
- Selector: reprezintă un obiect de tip multiplexor şi este punctul central al acestei tehnologii. Acesta monitorizează socket-urile înregistrate şi serializează cererile sosite de la acestea, transmiţându-le către aplicaţia server.
- Cheile reprezintă obiectele ce încapsulează cererile sosite de la clienţi.

Un algoritm general pentru a construi un server neblocant arată astfel:

```
create SocketChannel;
create Selector
associate the SocketChannel to the Selector
for(;;) {
  waiting events from the Selector;
  event arrived; create keys;
  for each key created by Selector {
    check the type of request;
    isAcceptable:
      get the client SocketChannel;
      associate that SocketChannel  to the Selector;
      record it for read/write operations
      continue;
```

```
   isReadable:
      get the client SocketChannel;
      read from the socket;
      continue;
   isWriteable:
      get the client SocketChannel;
      write on the socket;
      continue;
  }
}
```

Implementarea serverului constă într-o buclă infinită în cadrul căreia selectorul aşteaptă producerea de evenimente. În momentul în care un eveniment s-a produs şi o cheie a fost generată se verifică tipul acestei chei. Tipurile de chei posibile sunt:

- Acceptable – asociată evenimentului de cerere de conexiune de la un client
- Connectable – asociată evenimentului de acceptare de conexiune de către client
- Readable – citire de date
- Writable – scriere de date

Clasa ***Selector*** este responsabilă pentru menţinerea unui set de chei care pot fi active în timpul rulării programului server. În momentul în care un evenimet este generat de către un client, o cheie este construită.

```
Selector selector = Selector.open();
```

Pentru a demultiplexa datele şi a avea acces la evenimente trebuie construit un canal de comunicaţie care va trebui înregistrat în cadrul obiectului de tip selector. Fiecare canal de comunicaţie înregistrat va trebui să specifice tipul de evenimente de care este interesat.

```
ServerSocketChannel channel = ServerSocketChannel.open();
channel.configureBlocking(false);
InetAddress lh = InetAddress.getLocalHost();
InetSocketAddress isa = new InetSocketAddress(lh, port );
channel.socket().bind(isa);

SelectionKey acceptKey = channel.register( selector, SelectionKey.OP_ACCEPT
);
```

Un canal care citeşte şi scrie date va fi înregistrat în felul următor:

```
SelectionKey readWriteKey = channel.register( selector,
SelectionKey. OP_READ| SelectionKey. OP_WRITE );
```

Codul aplicaţiei server ce implementează comunicarea prin socket-uri nonblocante este listat mai jos:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
```

```
import java.nio.channels.spi.*;
import java.nio.charset.*;
import java.net.*;
import java.util.*;

public class NonBlockingServer2 {

      public static void main(String[] args) throws Exception{

//          Create the server socket channel
            ServerSocketChannel server = ServerSocketChannel.open();
//          nonblocking I/O
            server.configureBlocking(false);
//          host-port 8000
            server.socket().bind(new
java.net.InetSocketAddress("localhost",8000));
            System.out.println("Server waiting on port 8000");
//          Create the selector
            Selector selector = Selector.open();
//          Recording server to selector (type OP_ACCEPT)
            server.register(selector,SelectionKey.OP_ACCEPT);

//          Infinite server loop

            for(;;) {
            Thread.sleep(1000);
              // Waiting for events
              System.err.println("wait for event...");
              selector.select();

              // Get keys
              Set keys = selector.selectedKeys();
              Iterator i = keys.iterator();
              System.err.println("keys size="+keys.size());
              // For each keys...
              while(i.hasNext()) {

                  // Obtain the interest of the key
                SelectionKey key = (SelectionKey) i.next();

                // Remove the current key
                i.remove();


                // if isAccetable = true
                // then a client required a connection
                if (key.isAcceptable()) {
                  System.err.println("Key is of type acceptable");
                  // get client socket channel
                  SocketChannel client = server.accept();
                  // Non Blocking I/O
                  client.configureBlocking(false);
                  // recording to the selector (reading)
                  client.register(selector, SelectionKey.OP_READ);
                  continue;
                }
```

```
                // if isReadable = true
                // then the server is ready to read
                if (key.isReadable()) {
                  System.err.println("Key is of type readable");
                  SocketChannel client = (SocketChannel) key.channel();

                  // Read byte coming from the client
                  int BUFFER_SIZE = 32;
                  ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
                  try {
                    client.read(buffer);

                  }
                  catch (Exception e) {
                    // client is no longer active
                        client.close();

                    e.printStackTrace();
                    continue;
                  }

                  // Show bytes on the console
                  buffer.flip();
                  Charset charset=Charset.forName("ISO-8859-1");
                  CharsetDecoder decoder = charset.newDecoder();
                  CharBuffer charBuffer = decoder.decode(buffer);
                  System.out.println(charBuffer.toString());
                  continue;
                }
              }
                System.err.println("after while keys size="+keys.size());
              }
      }
}
```

Codul aplicaţiei client ce foloseşte socketu-uri neblocante pentru comunicarea cu un server este listat mai jos:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.channels.spi.*;
import java.nio.charset.*;
import java.net.*;
import java.util.*;

public class NonBlockingClient {
          public static void main(String[] args) throws IOException {

// Create client SocketChannel
            SocketChannel client = SocketChannel.open();

// nonblocking I/O
            client.configureBlocking(false);
```

```
// Connection to host port 8000
            client.connect(new java.net.InetSocketAddress("localhost",8000));

// Create selector
            Selector selector = Selector.open();

// Record to selector (OP_CONNECT type)
            SelectionKey
            clientKey = client.register(selector, SelectionKey.OP_CONNECT);

// Waiting for the connection
            while (selector.select(500)> 0) {
                  System.err.println("Start communication...");

                  // Get keys
                  Set keys = selector.selectedKeys();
                  Iterator i = keys.iterator();

                  // For each key...
                  while (i.hasNext()) {
                      SelectionKey key = (SelectionKey)i.next();

                      // Remove the current key
                      i.remove();

                      // Get the socket channel held by the key
                      SocketChannel channel = (SocketChannel)key.channel();

                       // Attempt a connection
                       if (key.isConnectable()) {

                       // Connection OK
                       System.out.println("Server Found");

                  // Close pendent connections
                  if (channel.isConnectionPending())
                    channel.finishConnect();

                  // Write continuously on the buffer
                  ByteBuffer buffer = null;
                  int x=0;
                  for (;x<7;) {
                        x++;
                    buffer =
                      ByteBuffer.wrap(
                        new String(" Client " + x + " "+x).getBytes());
                    channel.write(buffer);
                    buffer.clear();
                    try {Thread.sleep(2000);}
                    catch (InterruptedException e) {e.printStackTrace();}
                  }
                  channel.finishConnect();
                  client.close();
                }
              }
```

```
            }
            System.err.println("Client terminated.");
        }
}
```

Exerciţii

1)

- Lansaţi în execuţie aplicaţia *ServerSimplu*
- Fără a opri aplicaţia *ServerSimplu* lansaţi în execuţie aplicaţia *ClientSimplu*

2)

- Opriţi aplicaţiile *ServerSimplu* şi *ClientSimplu*
- Lansaţi în execuţie aplicaţia *ServerMultifir*
- Fără a opri aplicaţia *ServerMultifir* lansaţi două sau mai multe instanţe ale aplicaţiei *ClientSimplu*
-

3) Testati aplicatia NonBlockingServer2