6. Programare distribuită cu socketuri

Programarea distribuită în rețea implică trimiterea de mesaje și date între aplicații ce rulează pe calculatoare aflate într-o rețea locală sau conectate la Internet.Pachetul care oferă suport pentru realizarea aplicațiilor de rețea în Java este java.net, iar clasele din acest pachet oferă o modalitate facilă de programare în rețea a unei aplicații distribuite.

6.1. Obiective

- Familiarizarea cu conceptele de bază protocol,port, adresă IP, socket utilizate pentru modelarea interacțiunii între procese distribuite
- Studiul bibliotecii Java.net
- Crearea unor aplicații distribuite în rețea

6.2. Concepte

Cele mai importante concepte specifice mecanismelor de tip socket existente în pachetul Java.net, referă câteva noțiuni fundamentale din domeniul rețelelor de calculatoare cum ar fi: protocol, port, adresa IP, socket , noțiuni ce vor fi definite în continuare.

Un *protocol* reprezintă o convenție de reprezentare a datelor folosită în comunicarea între două calculatoare. Având ın vedere faptul ca orice informație care trebuie trimisă prin rețea trebuie serializată astfel încât să poată fi transmisă secvențial, octet cu octet către destinație, era nevoie de stabilirea unor convenții referitor la transmiterea de mesaje care să fie folosite atât de calculatorul care trimite datele cât și de cel care le primește, pentru a putea comunica. Două dintre cele mai utilizate protocoale în rețea sunt TCP și UDP, caracterizate de următoarele aspecte:

• *TCP (Transport Control Protocol)* este un protocol ce furnizeaza un flux sigur de date între doua calculatoare aflate în rețea și asigură stabilirea unei conexiuni permanente între cele două calculatoare pe parcursul comunicației.
• *UDP (User Datagram Protocol)* este un protocol bazat pe pachete independente de date, numite datagrame, trimise de la un calculator către altul fără a se garanta în vreun fel ajungerea acestora la destinație sau ordinea în care acestea ajung. Acest protocol nu stabilește o conexiune permanentă între cele două calculatoare.

*Adresa IP.*Orice calculator conectat la Internet este identificat în mod unic de adresa sa IP (IP este acronimul de la Internet Protocol). Aceasta reprezintă un număr pe 32 de biți, uzual sub forma a 4 octeți(IPv4), cum ar fi de exemplu: 193.231.30.131 și este numit adresa IP numerică. Corespunzătoare unei adrese numerice există și o adresa IP simbolică, pentru adresa numerică anterioară. De asemenea, fiecare calculator aflat într-o rețea locală are un nume unic ce poate fi folosit la identificarea acestuia.

Noțiunea de port. Un calculator are în general o singură legatură fizică la rețea. Orice informație destinată unei anumite mașini trebuie deci să specifice obligatoriu adresa IP a acelei mașini. Însă pe un calculator pot exista concurent mai multe procese care au stabilite conexiuni în rețea, asteptând diverse informații. Prin urmare, datele trimise către o destinație trebuie să specifice pe lângă adresa IP a calculatorului și *procesul* către care se îndreaptă informațiile respective. Identificarea proceselor se realizează prin intermediul porturilor.*Un port este un număr pe 16 biți care identifică în mod unic procesele care rulează pe o anumită mașină*. Orice aplicație care realizează o conexiune în rețea va trebui să atașeze un număr de port acelei conexiuni. Valorile pe care le poate lua un număr de port sunt cuprinse ıntre 0 și 65535 (deoarece sunt numere reprezentate pe 16 biți), numerele cuprinse între 0 si 1023 fiind însă rezervate unor servicii sistem și din acest motiv nu trebuie folosite în aplicații.

Pentru implementarea aplicaţiilor de reţea pachetul *Java.net* oferă suport independent de sistem, având la baza modelului comunicaţia de tip client - server. Acest pachet conţine clase, interfeţe şi excepţii ce implementează în formă obiectuală conceptele uzuale de comunicare. Astfel, un client Java, pentru a obţine un anumit serviciu de la un server remote, foloseşte adresa serverului pentru a solicita conexiunea, iar serverul ascultă reţeaua şi aşteaptă cereri din partea clienţilor. Stabilirea conexiunii presupune utilizarea ca şi identificatori de hosturi comunicante a adresei IP şi a adresei portului.

6.2.1. Elemente în programarea cu socketuri

Pentru a permite lucrul cu identificatori simbolici de tip nume dar şi cu adresele IP ale hosturilor din reţea, în Java este definită clasa InetAddress ce furnizează abstracţiunile necesare accesării hosturilor.

Metodele statice ale clasei sunt:

- *Public static InetAddress getByName(String host) throws UnknownHostException* returnează numele hostului local pentru care se cunoaşte adresa IP.
- *Public static InetAddress getLocalhost() throws UnknownHostException*-determină adresa IP a hostului local
- *Public static InetAddress getAllByName(String host)* - returnează în tablou adreseleIP ale hostului curent
- *Public byte getAddress(),* returnează adresa IP a obiectului curent în formă de şir
- *Public String getHostName()* returnează numele calculatorului căruia îi corespunde obiectul InetAddress
- *Public String getHostAddress()* – returnează adresa IP căruia îi corespunde obiectul InetAddress

Clasa URL oferă facilităţi necesare manipulării locatorilor universali de resurse URL.

Forma completă a unui asemenea locator este: *Protocol://nume_calculator:port/nume_cale#ref*

Numele calculatorului poate fi în format simbolic sau adresă Internet, portul se specifică dacă nu este folosit cel standard, iar ref nu este parte a URL ci referă o parte din documentul specificat funcţie de tipul resursei.Clasa care permite lucrul cu URL-uri este java.net.URL. Aceasta are mai mulți constructori pentru crearea de obiecte ce reprezintă referințe către resurse aflate în rețea, cel mai uzual fiind cel care primește ca parametru un șir de caractere. In cazul în care șirul nu reprezintă un URL valid va fi aruncată o excepție de tipul MalformedURLException.

Constructorii clasei sunt:

- *Public URL (String protocol, String host, int port, String file) throws MalformedURLException* – creează un obiect URL
- *Public URL (String protocol, String host, String file) throws MalformedURLException*, crează un nou obiect URL absolut pentru care se foloseşte portul standard
- *Public URL (URL context, String spec) throws MalformedURLException-* creează un URL bazat pe interpretarea specificării spec corespunzător contextului dat., dacă argumentul context este null, iar spec este doar o specificare parţială componentele lipsă vor fi preluate din context.
- *Public URL (String spec) throws MAlformedURLException* – ceează un obiect URL din reprezentarea string dată.

Un obiect de tip URL poate fi folosit pentru:

- Aflarea informațiilor despre resursa referită (numele calculatorului gazdă, numele fișierului, protocolul folosit. etc).

- Citirea printr-un flux a conținutului fișierului respectiv.
- Conectarea la acel URL pentru citirea și scrierea de informații.

Conectarea la un URL se realizează prin metoda openConnection ce stabilește o conexiune bidirecțională cu resursa specificată. Aceasta conexiune este reprezentată de un obiect de tip URLConnection, ce permite crearea atât a unui flux de intrare pentru citirea informațiilor de la URL-ul specificat, cât și a unui flux de ieșire pentru scrierea de date către acel URL.

Operațiunea de trimitere de date dintr-un program catre un URL este similară cu trimiterea de date dintr-un formular de tip FORM aflat ıntr-o pagina HTML. Metoda folosită pentru trimitere este POST. In cazul trimiterii de date, obiectul URL este uzual un proces ce rulează pe serverul Web referit prin URL-ul respectiv (jsp, servlet, cgi-bin, php, etc).

6.2.2.Socketuri TCP

Un socket (soclu) este o abstracțiune software folosită pentru a reprezenta fiecare din cele două ”capete” ale unei conexiuni între doua procese ce rulează într-o rețea. Fiecare socket este atașat unui port astfel încât să poata identifica unic procesul (aplicația) căruia îi sunt destinate datele.

Socket-urile sunt de două tipuri:

1. TCP, implementate de clasele *Socket si ServerSocket;*
2. UDP, implementate de clasa *DatagramSocket.*

O aplicație de rețea ce folosește socket-uri se încadrează în modelul client/server de concepere a unei aplicații. In acest model aplicația este formată din două categorii distincte de programe numite servere, respectiv clienți.Programele de tip server sunt cele care oferă diverse servicii eventualilor clienți, fiind în stare de așteptare atât timp cât nici un client nu le solicită serviciile. Programele de tip client sunt cele care initiază conversația cu un server, solicitand un anumit serviciu. Uzual, un server trebuie sa fie capabil să trateze mai mulți clienți simultan și, din acest motiv, fiecare cerere adresată serverului va fi tratată într-un fir de execuție separat.

Incepând cu versiunea 1.4 a platformei standard Java, exista o clasă utilitară care implementează o pereche de tipul (adresa IP, numar port). Aceasta este *InetSocketAddress (*derivata din SocketAddress), obiectele sale fiind utilizate de constructori și metode definite ın cadrul claselor ce descriu socketuri, pentru a specifica cei doi parametri necesari identificării unui process care trimite sau receptionează date în rețea.

Clasa Socket .Clasa *Socket (*din pachetul java.net) oferă posibilitatea comunicării folosind socketurile, iar clasa *ServerSocket* implementează un socket folosit în servere pentru a aştepta conexiuni de la clienţi. Un socket în Java este reprezentarea unei conexiuni TCP, folosind această clasă un client poate crea canale de comunicaţie de tip streamuri de date. După stabilirea conexiunii, hosturile local, respectiv remote pot comunica folosind o conexiune tip full duplex.

Constructorii cei mai utilizaţi ai acestei clase sunt:

- *Protected Socket ()* creează un socket neconectat,care poate fi ulterior ataşat la o conexiune.
- *Socket (String host, int port) throws UnknownHostException, IOException* - creează un socket TCP şi încearcă conectarea la portul hostului specificat prin *nume.*
- *Socket (InetAddress address, int port) throws IOException–* creează un socket TCP şi-l conectează la portul specificat al hostului identificat prin *adresă.*

Socket (String host, int port, Inet Address localAddr, int localPort) throws IOException şi *Socket (InetAddress address, int port, InetAddress localAddr, int local Port) throws IOException,*creează socketul TCP, îl leagă la hostul (adresa) şi portul local şi se conectează la hostul remote. Acesta este constructorul care permite alegerea manuală a interfeţei care va fi conectată.

Metodele clasei permit identificarea hostului remote, a porturilor utilizate şi extragerea

streamurilor utilizate în comunicaţia bidirecţională. Comunicaţia folosind socketurile în Java presupune crearea unui Socket apoi utilizarea metodelor *getInputStream()* şi *getOutputStream()* pentru a obţine streamurile necesare comunicaţiei. Alte metode definite în această clasă sunt: *getInetAddress(), getPort(), toString().*Excepţiile sunt cele cauzate de imposibilitatea legării la hostul local sau remote, refuzarea unei conexiuni datorită inexistenţei serverului care să asculte pe un anumit port sau imposibilitatea atingerii hostului remote.

Clasa ServerSocket.Această clasă oferă funcţionalitatea mecanismului prin care un server *acceptă conexiuni* de la clienţi, ea permite crearea unei conexiuni socket pentru fiecare client apoi serverul va gestiona aceste conexiuni extrăgând streamuri de intrare şi ieşire pentru a comunica cu clientul, după ce a fost creată o conexiune de acest tip, va putea fi utilizat un socket pentru a transfera date.

Ciclul de viaţă al unui server clasic conţine următoarele etape: crearea unui nou socket- server care va asculta conexiuni folosind metoda accept() (metoda care va returna un obiect de tip socket), efectuarea de transferuri de streamuri (in, out sau in şi out), clientul şi serverul vor continua interacţiunea respectînd un anumit protocol, apoi se va închide conexiunea de către unul din capetele comunicante şi în final se va aştepta după o nouă conexiune.Proiectarea unui server complex presupune utilizarea threadurilor astfel încât serverul să fie capabil să proceseze cât de repede posibil orice nouă conexiune.

Constructorii clasei pentru crearea de noi socketuri sunt:

- *ServerSocket(int port) throws IOException, BindException*– creează un socket la portul specificat, valoarea zero semnifică orice port disponibil.
- *ServerSocket(int port, int backlog) throws IOException, BindException* – creează un socket la portul specificat şi cu numărul de cereri de conectare care pot fi păstrate în aşteptare (înainte de a refuza alte cereri, implicit 50 cereri).
- *ServerSocket (int port, int backlog, InetAddress bindAddr) throws IOException*– socketul creat se leagă la adresa de IP specificată. Este utilă pentru sistemele cu mai multe adrese de IP, permiţând alegerea acesteia.

Complementar există definiţi constructori care pot fi utilizaţi pentru a crea propriile implementări de socketuri, eventual utilizând servere de proxy sau diverse protocoale de securitate.

Metodele clasei

- *accept()throws IOException*- metoda este utilizată la acceptarea unei conexiuni. Este soluţia de comunicaţie blocantă deoarece serverul va fi oprit până la conectarea clientului.
- *getInetAddress()* – returnează adresa utilizată de server (a hostului local, pentru sisteme multihomed nu se poate previziona care adresă va fi returnată)
- *getLocalPort()* – permite ascultarea unui port nespecificat, această metodă permite aflarea acestui port.

Inchiderea unui socket

Multe din protocoale folosesc presupunerea că aceste socketuri se închid singure după un anumit număr de schimburi de mesaje deoarece pentru anumite programe care pot rula la infinit este necesar ca socketurile să fie închise.

Close() throws IOException – eliberează portul pentru alte programe care îl pot utiliza, prin închiderea conexiunii.

După închiderea unui socket, numărul de port şi adresa locală sunt încă accesibile folosind metodele *GetInetAddress(), getLocalPort(), getPort().*

Setarea opţiunilor specifice unui socket- (set)getSoTimeout(int timeout) - sunt utilizate în protocoale complexe care necesită conexiuni multiple între client şi server (valoarea implicită este 0,

de obicei setarea parametrului presupune specificarea înaintea metodei accept() a valorii de timeout). Alţi parametri care este necesar să fie setaţi în cazul unor protocoale complexe sunt *TCP_Nodelay* – asigură transmiterea pachetelor cât de repede posibil sau *SO_Linger* – specifică modul de tratare a datagramelor transmise după închiderea unui socket.

In acest model se stabileşte o conexiune TCP între o aplicaţie client şi o aplicaţie server care furnizează un anumit serviciu. Avantajul protocolul TCP/IP este că asigură realizarea unei comunicări stabile, permanente în reţea, existând siguranţa că informaţiile trimise de un proces vor fi recepţionate corect şi complet la destinaţie sau va fi semnalată o excepţie în caz contrar.

Legătura între un client şi un server se realizeaza prin intermediul a două obiecte de tip Socket, câte unul pentru fiecare capat al "canalului" de comunicaţie dintre cei doi. La nivelul clientului crearea socketului se realizează specificând adresa IP a serverului şi portul la care rulează acesta, constructorul uzual folosit fiind: Socket(InetAddress address, int port).

La nivelul serverului, acesta trebuie să creeze întâi un obiect de tip ServerSocket. Acest tip de socket nu asigură comunicarea efectivă cu clienţii ci este responsabil cu "ascultarea" reţelei şi crearea unor obiecte de tip Socket pentru fiecare cerere aparută, prin intermediul căruia va fi realizată legătura cu clientul. Crearea unui obiect de tip ServerSocket se face specificând portul la care rulează serverul, constructorul folosit fiind: ServerSocket(int port).

Metoda clasei ServerSocket care aşteaptă "ascultă" reteaua este *accept()*. Aceasta blochează procesul părinte până la apariţia unui cereri şi returnează un nou obiect de tip Socket ce va asigura comunicarea cu clientul.

Blocarea poate să nu fie permanentă ci doar pentru o anumită perioada de timp - aceasta va fi specificată prin metoda setSoTimeout, cu argumentul în milisecunde.

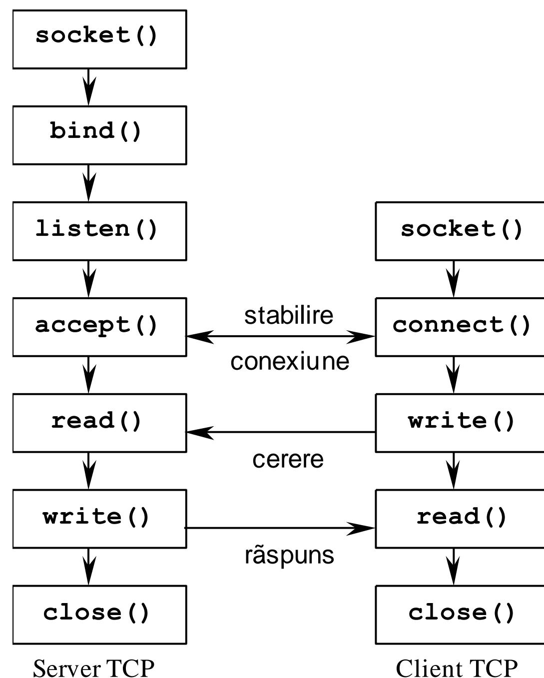

Figura 6.1. Mecanismul socket TCP

Pentru fiecare din cele două socketuri deschise pot fi create apoi două fluxuri pe octeti pentru citirea, respectiv scrierea datelor. Acest lucru se realizează prin intermediul metodelor *getInputStream,* respectiv *getOutputStream*.

Fluxurile obţinute vor fi folosite împreună cu fluxuri de procesare care să asigure o comunicare facilă între cele două procese. In funcţie de specificul aplicaţiei acestea pot fi perechile:

- *BufferedReader, BufferedWriter ¸si PrintWriter* - pentru comunicare prin intermediul sirurilor de caractere;
- *DataInputStream, DataOutputStream* - pentru comunicare prin date primitive;
- *ObjectInputStream, ObjectOutputStream* - pentru comunicare prin intermediul obiectelor;

Structura generală a unui server bazat pe conexiuni este următoarea:

1. Creează un obiect de tip ServerSocket la un anumit port
2. Asteaptă realizarea unei conexiuni cu un client, folosind metoda accept; (va fi creat un obiect nou de tip Socket)
3. Tratează cererea venită de la client:
   3.1 Deschide un flux de intrare şi primeşte cererea
   3.2 Deschide un flux de ieşire şi trimite răspunsul
   3.3 Inchide fluxurile şi socketul nou creat

Este recomandat ca tratarea cererilor să se realizeze în fire de execuție separate, pentru ca metoda accept să poata fi reapelată cât mai repede în vederea stabilirii conexiunii cu un alt client.

Structura generală a unui client bazat pe conexiuni este:

1. Citeste sau declară adresa IP a serverului și portul la care acesta rulează;
2. Creează un obiect de tip Socket cu adresa și portul specificate;
3. Comunică cu serverul:
    3.1 Deschide un flux de ieșire și trimite cererea;
    3.2 Deschide un flux de intrare și primeste răspunsul;
    3.3 Inchide fluxurile și socketul creat;

6.2.3. Socketuri datagrame

In acest model nu există o conexiune permanentă între client și server prin intermediul căreia să se realizeze comunicarea. Clientul trimite cererea către server prin intermediul unuia sau mai multor pachete de date independente, serverul le receptionează, extrage informațiile conținute și returnează răspunsul tot prin intermediul pachetelor. Un astfel de pachet se numește datagramă și este reprezentat printr-un obiect din clasa *DatagramPacket*. Rutarea datagramelor de la o mașina la alta se face exclusiv pe baza informațiilor conținute de acestea. Primirea și trimiterea datagramelor se realizează prin intermediul unui socket, modelat prin intermediul clasei *DatagramSocket.*

După cum s-a mentionat, dezavantajul acestei metode este că nu garantează ajungerea la destinație a pachetelor trimise și nici că vor fi primite în aceeaşi ordine în care au fost expediate. Pe de altă parte, există situații în care aceste lucruri nu sunt importante și acest model este de preferat celui bazat pe conexiuni care solicită mult mai mult atât serverul cât și clientul. De fapt, protocolul TCP/IP foloseşte tot pachete pentru trimiterea informațiilor dintr-un nod în altul al rețelei, cu deosebirea că asigură respectarea ordinii de transmitere a mesajelor și verifica ajungerea la destinație a tuturor pachetelor în cazul în care unul nu a ajuns, acesta va fi retrimis automat.

In pachetul Java.net există clasele *DatagramSocket, DatagramPacket şi MulticastSocket* care oferă suport pentru programarea transmisiei de date folosind datagrame. Fiecare datagramă conţine un antet (header) şi o zonă de date. Antetul cuprinde informaţiile de adresare (portul şi adresa sursă şi destinaţie) şi alte informaţii legate de asigurarea transmisiei. Datagramele au lungime fixă de aceea este nevoie uneori să fie împărţite în pachete şi refăcute la destinaţie. Transmisia de datagrame se realizează fără stabilirea prealabilă a unei conexiuni, pachetele transmise nu trebuie recepţionate în aceeaşi ordine şi deasemenea un socket de tip datagramă poate recepţiona date de la mai multe hosturi diferite.

Clasa DatagramPacket

Clasa DatagramPacket organizează datele în pachete tip UDP. Constructorii clasei DatagramPacket sunt:

- *DatagramPacket (byte buffer [], int length)-* unde, buffer reprezintă un tablou care memorează pachetul, iar length reprezintă numărul maxim de bytes care pot fi citiţi în buffer.
- *DatagramPacket (byte buffer [], int length, Inetaddress address, int port)* – pachetul va fi livrat la adresa de host şi portul specificate. Este necesar ca la celălalt capăt al comunicaţiei să existe un server UDP care să asculte conexiuni.Volumul de date care poate fi plasat într-un pachet depinde de tipul protocolului, şi de tipul tehnologiei utilizate.

Metodele acestei clase sunt:

- *GetAddress()* – returnează obiectul InetAddress reprezentând adresa hostului remote.
- *GetPort()* - returnează portul datagramei pe maşina remote.
- *GetLength()* - returnează lungimea datagramei (număr de bytes)
- *GetData()* – returnează un tablou care conţine datele transferate folosind datagrame.

Clasa DatagramSocket
Această clasă permite transmisia şi recepţia datagramelor. Constructorii clasei sunt:

- *DatagramSocket() throws SocketException* - creează un socket datagramă la un port aleator ales (anonim).
- *DatagramSocket (int port) throws SocketException* - creează un socket care ascultă la un port specificat.
- *DatagramSocket (int port, InetAddress local) throws SocketException* – socketul creat este şi legat la o adresă locală.

Transmisia şi recepţia datagramelor

După crearea unui DatagramPacket şi construirea unui DatagramSocket pachetul acesta va fi transferat folosind metoda send(), respectiv recepţionat folosind metoda receive().

send (DatagramPacket dp) throws IOException
receive (DatagramPacket dp) throws IOException

Dupa crearea unui pachet procesul de trimitere și primire a acestuia implică apelul metodelor send și receive ale clasei *DatagramSocket.* Deoarece toate informațiile sunt incluse în datagramă, același socket poate fi folosit atât pentru trimiterea de pachete, eventual către destinații diferite, cât și pentru recepționarea acestora de la diverse surse.

In cazul în care refolosim pachete, putem schimba conținutul acestora cu metoda *setData*, precum și adresa la care le trimitem prin *setAddress, setPort* si *setSocketAddress.* Extragerea informațiilor conținute de un pachet se realizează prin metoda getData din clasa DatagramPacket. De asemenea, această clasă oferă metode pentru aflarea adresei IP și a portului procesului care a trimis datagrama, pentru a-i putea răspunde dacă este necesar.

Acestea sunt: *getAdress, getPort si getSocketAddress.*

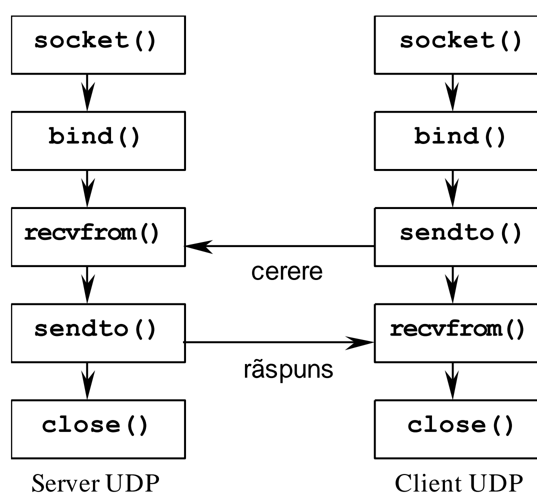

Figura 6.2. Mecanismul socket datagrame

6.2.4.Socketuri multicast

Transmisia (recepţia) multicast oferă un set de avantaje de loc de neglijat între care simplificarea proiectării şi reducerea consumului de lăţime de bandă pentru un set de aplicaţii a căror topologie necesită arbori de acoperire (din domeniul procesării masiv paralele, a serviciilor de nume, a serviciilor de directoare distribuite, a mecanismelor de replicare a bazelor de date, etc.).

Pentru a recepţiona date multicast de la un host remote, după crearea unui socket multicast este necesară asocierea la un grup de multicast apoi va putea fi stabilită conexiunea pentru a transmite şi recepţiona date. Asocierea la grup anunţă ruterele aflate în calea către serverul care oferă serviciul să transmită datele în mod multicast, hostul fiind responsabil cu adresarea pachetelor în mod multicast. Ataşarea la grup determină comunicaţia similară celei utilizate la transferul de datagrame, trebuie remarcat faptul că doar pentru a transmite în mod multicast nu este necesară ataşarea la un grup multicast.

Ceea ce diferenţiază totuşi comunicarea multicast de cea unicast o constituie specificarea şi gestionarea câmpului TTL (Time to live) câmp ce permite evitarea buclelor de rutare pentru topologii arborescente complexe.Adresele de tip multicast sunt adrese de clasă D în gama de adrese 224.0.0.0 – 239.255.255.255.255, domeniu structurat pe diferite tipuri de adrese multicast rezervate sau nu.

Constructorii clasei *MulticastSocket ()* utilizaţi în procesarea datagramelor multicast sunt:
MulticastSocket() throws SocketException – crează un socket legat la portul anonim (orice port asignat de sistem) şi util pentru clienţii ce nu necesită specificarea unui port.
MulticastSocket(int port) throws SocketException

Metodele clasei necesare comunicării cu un grup multicast sunt:

- *JoinGroup(Inetaddress mcastaddr) throws SocketException*
- *LeaveGroup(Inetaddress mcastaddr) throws SocketException*
- *Send (DatagramPacket dp, byte ttl) throws IOException, SocketException*
- *Set (Get)Interface (InetAddress interface) throws SocketException* – pentru alegerea interfeţei utilizate pentru transmisia multicast în hosturile multihomed (cu mai multe adrese IP).

Pentru recepţia de pachete multicast poate fi folosită metoda receive a claselor DatagramSocket.

6.3.Exemple

6.3.1. Implementarea interacțiunii dintre un client și un server folosind java.net , având drept funcționalitate trasmiterea unui fișier(sursa [4]).

Clientul

```
// Get a file from RemoteFileServer and print it on stdout.
//  usage:javac Client.java
//    java Client "hostname" "filename"  (after starting the server)

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

public class Client {
  public static void main(String[] args) {
    try {
      // read command-line arguments
      if (args.length != 2) {
        System.out.println("need 2 arguments");
        System.exit(1);
      }
      String host = args[0];
      String filename = args[1];

      // open socket, then input and output streams to it
      Socket socket = new Socket(host,9999);
      BufferedReader from_server =
        new BufferedReader(new InputStreamReader(socket.getInputStream()));
      PrintWriter to_server = new PrintWriter(socket.getOutputStream());

      // send filename to server, then read and print lines until
      // the server closes the connection
      to_server.println(filename); to_server.flush();
      String line;
      while ((line = from_server.readLine()) != null) {
        System.out.println(line);}}
    catch (Exception e) {    // report any exceptions
      System.err.println(e);}}}
```

Serverul

```
// Simple server that reads a file and sends it back to a client.
// usage:
//   javac FileReaderServer.java
//   java FileReaderServer

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

public class FileReaderServer {
  public static void main(String args[]) {
    try {
      // create server socket to listen for connection on port 9999
      ServerSocket listen = new ServerSocket(9999);

      while (true) {
        System.out.println("server waiting for connection");
        Socket socket = listen.accept();  // wait for a client
        // create input and output streams to talk to client
        BufferedReader from_client =
          new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter to_client = new PrintWriter(socket.getOutputStream());

        // get filename from client and check if it exists
        String filename = from_client.readLine();
        File inputFile = new File(filename);
        if (!inputFile.exists()) {
          to_client.println("cannot open " + filename);
          to_client.close(); from_client.close();
          socket.close();
          continue;}

        // read lines from filename and send them to the client
        System.out.println("reading from file " + filename);
        BufferedReader input =
          new BufferedReader(new FileReader(inputFile));
        String line;
        while ((line = input.readLine()) != null)
          to_client.println(line);
        to_client.close(); from_client.close();
        socket.close();}}
    catch (Exception e) {  // report any exceptions
      System.err.println(e);}}}
```

6.3.2. Se vor testa exemplele ce ilustrează modul de utilizare a API-ului MPI, exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/06_Socketuri.

6.4.Întrebări teoretice

6.4.1. Care este diferența dintre un socket și un port? Ce reprezintă protocolul
6.4.2.Ce tipuri de socketuri sunt implementate în pachetul Java.net? identificați cele mai importante două elemente distinctive.
6.4.3.Ce aplicabilitate pot avea socketurile multicast? Descrieți un miniscenariu de aplicație.

6.5.Probleme propuse

6.5.1.Proiecaţi o aplicaţie care să implementeze comunicaţia între un client şi un server respectând specificaţiile: programul server ascultă reţeaua şi aşteaptă să primească o cerere de la un client. Dacă cererea este recepţionată fără conflicte, serverul procesează cererea şi închide conexiunea. Clientul foloseşte o adresă pentru a solicita o conexiune unui calculator în reţea (în mod uzual solicită un serviciu unui server), apoi se deconectează din reţea.

6.5.2.Propuneti o soluţie de server neblocant care să utilizeze valori de timeout. Proiectaţi un server concurent care să genereze pentru fiecare conexiune acceptată un nou fir de execuţie.

6.5.3.Să se implementeze o aplicaşie de tip client-server în care pentru a accesa serverul, acesta solicită o parolă pentru autentificarea unui client apoi va trimite în reţea un fişier către programul client. Programul client se va conecta la server, va trimite parola solicitată, va solicita un fişier şi apoi va salva acel fișier.

6.5.4.Să se implementeze o aplicație de tip chat în arhitectură client-server fără persistarea informațiilor specifice .

6.6.Referințe bibliografice

1. Tutorial Oracle socketuri :https://docs.oracle.com/javase/tutorial/networking/sockets/
2. Tutorial practic socketuri :http://csis.pace.edu/~marchese/CS865/Lectures/Liu4/sockets
3. API : https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html
4. Socketuri java în aplicații :http://csis.pace.edu/~marchese/CS865/Lectures/Liu4/Javasockets.pdf