5. Elemente de programare Java pentru retele bazate pe IP

5.1. Introducere in Protocolul Internet (IP) si stiva de protocoale IP

5.1.1. Elemente de terminologie a retelor de comunicatie

O **retea de comunicatie** (*communication network*) este un sistem de comunicatie de forma unui graf format din sisteme intermediare cu rol de retransmisie (noduri interne) si sisteme terminale cu rol de transmitator-receptor (noduri de capat) interconectate.

O **retea de calculatoare** (*computer network*) este o varianta de sistem de comunicatie in care sistemele terminale sunt **calculatoare** (numite **masini, gazde** sau ***host-uri***).

O retea de calculatoare de arie geografica mare, **WAN** (*Wide Area Network*), interconecteaza calculatoare la nivel national sau global.
LAN (*Local Area Network*) = retea locala de calculatoare (interconecteaza calculatoare aflate la cel mult cateva zeci de metri distanta). Vitezele (debitele) de transmisie tipice in retele locale sunt 10-100 Mb/s, desi pot fi intalnite si retele la 1 Gb/s si peste.

Ethernetul este o tehnologie de retele locale cu topologie de tip bus (magistrala, cu acces concurent).

Un ***internet*** este o retea formata din (sub)retele interconectate.

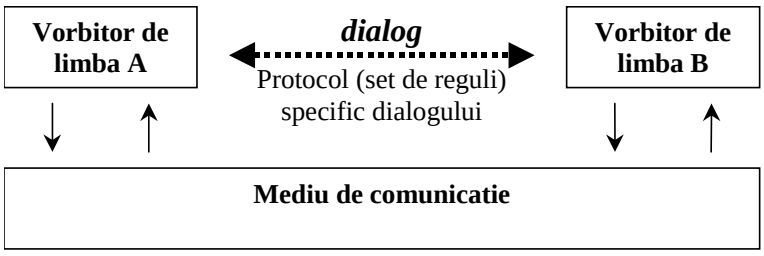
Un ***gateway*** este un element *hardware* sau *software* prin care se realizeaza conectarea subretelelor unui internet.

Internetul este reteaua de calculatoare globala.

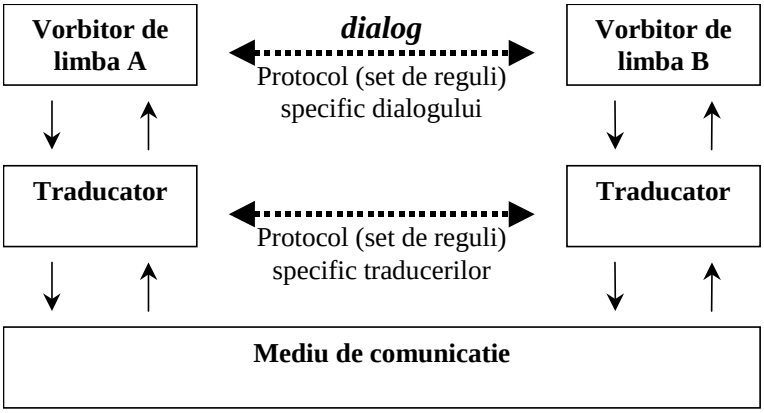
5.1.2. Modele de comunicatie stratificata

Pentru a ilustra modelul de comunicatie stratificata, bazat pe protocoale de comunicatie, consideram exemplul unui **dialog la distanta intre interlocutori de limbi diferite.**

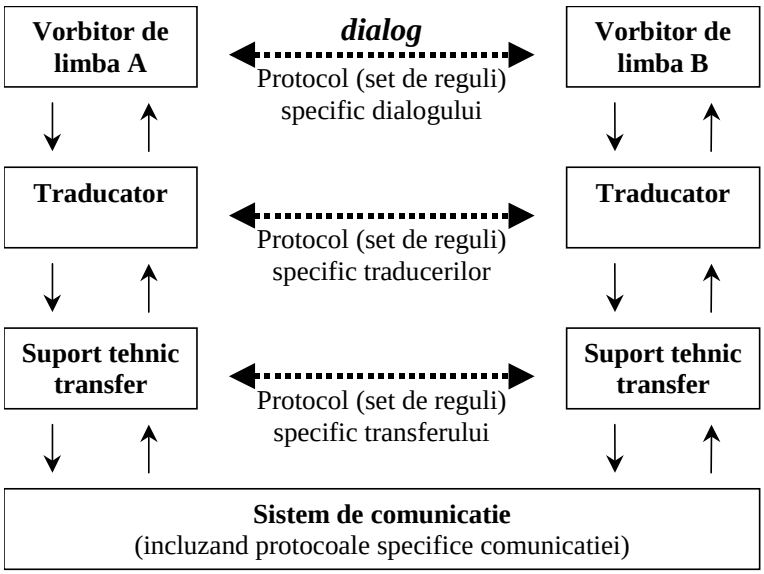
La nivelul cel mai inalt, dialogul poate fi modelat astfel:



Detaliind cazul in care **asigurarea suportului comun** (a unei limbi comune, engleza de exemplu) **prin utilizarea unor traducatori**:



Detaliind cazul in care **comunicatia la distanta include suport tehnic local si sistem de comunicatie**:



Un sistem de comunicatie stratificat utilizeaza acelasi model, bazat pe **niveluri** si **protocoale** de comunicatie.

Un **protocol** de comunicatie este un **set de reguli si conventii** care au fost **convenite intre participantii la o comunicatie** pentru a **asigura buna desfasurare a comunicatiei respective**.

Intr-un sistem de comunicatie stratificat, **interactiunile pe orizontala** se desfasoara **conform protocolului** de la respectivul nivel, interactiuni **virtuale**, realizate prin intermediul interactiunilor de la nivelurile inferioare.

Interactiunile pe verticala cu nivelul inferior au rolul de a transmite spre acest nivel **sarcini** cu scopul transferului mesajelor catre partenerul de la acelasi nivel, si de a prelua de la acest nivel **mesajele** provenite de la partenerul de la acelasi nivel.

Interactiunile pe verticala cu nivelul superior au rolul de a prelua de la acest nivel a **sarcinilor** necesare transferului mesajelor catre partenerul sau (de nivel superior), si de a transmite acestui nivel **mesajele** provenite de la partenerul sau (de nivel superior).

Un serviciu este un **set de functii (sarcini) posibil de oferit de catre nivelul inferior**, respectiv **setul de sarcini posibil de solicitat de catre nivelul superior**, in interactiunile **pe verticala**.

Protocolul este independent de continutul mesajelor schimbate in timpul comunicatiei. **Regulile pot fi diferite pentru sursa** si respectiv **destinatia** mesajelor.

Complexitatea inerenta comunicatiei este **mai usor implementata, controlata, gestionata** prin **separarea in niveluri ierarhice**.

Dualitatea **protocol-serviciu**, ca si dualitatea **implementare-interfata** din programarea bazata pe obiecte, provin din **principiul separarii preocuparilor** (***concern separation***, detaliere a principiului ***divide et impera***).

Esenta acestui principiu este: **simplificarea** (controlului) **prin separarea specificatiei functionale** (setul de functii, interfata, serviciul) **de realizarea propriu-zisa** (protocolul, implementarea).

Prin **aplicarea repetata** a acestui principiu:
- se realizeaza **descompuneri** ierarhice sau obiectuale ale caror **componente** (protocoale sau obiecte) sunt **mai usor de realizat si controlat**,
- specificatiile functionale (serviciile sau interfetele) pot fi realizate in diferite moduri (sunt **independente** de implementari),
- se obtine o **specializare modulara** a nivelurilor ierarhice respectiv obiectelor.

5.1.3. Modelul de interconectare a sistemelor deschise (modelul OSI)

Modelul OSI cuprinde **7 niveluri** ierarhice:

- **nivelul aplicatie (7)** - asigura interfete comune diferitelor aplicatii oferite utilizatorilor
- **nivelul prezentare (6)** - asigura sintaxe comune intre aplicatii sau utilizatori
- **nivelul sesiune (5)** - asigura gestiunea dialogului intre aplicatii sau utilizatori
- **nivelul transport (4)** - asigura diferite clase de transfer cap-la-cap (intre sistemele terminale utilizate de aplicatii)
- **nivelul retea (3)** - asigura transferul cap-la-cap (intre sistemele terminale ale retelei)
- **nivelul legatura de date (2)** - asigura transferul intre nodurile intermediare ale retelei
- **nivelul fizic (1)** - specifica parametrii electrici si mecanici ai comunicatiei

Pot fi identificate urmatoarele **subsisteme ierarhice**:

- **superior (subsistemul aplicatie)** – cuprinde nivelurile 7..4 OSI, include aspectele direct legate de aplicatie
- **inferior (subsistemul retea)** – cuprinde nivelurile 3..1 OSI, include aspectele direct legate de reteaua de comunicatie

Rolul nivelului transport:

- **ascunde detaliile aplicatiei** pentru retea (comunicatia propriu-zisa)
- **ascunde detaliile retelei** (comunicatiei propriu-zise) pentru aplicatie
- **cerintele specificate de aplicatie** si **oferta specificata pentru retea** sunt tratate similar (aduse la un numitor comun) la acest nivel, prin intermediul **parametrilor de calitate a serviciilor**

5.1.4. Modelul de comunicatie si protocoalele Internet

Modelul de comunicatie Internet cuprinde **4 niveluri** ierarhice:

- **aplicatie** (4 = 7..5 OSI) - corespunde subsistemului aplicatie OSI
- **transport** (3 = 4 OSI) - corespunde nivelului transport OSI
- **retea** (2 = 3 OSI) - corespunde nivelului retea OSI
- **interfata (acces) retea** (1 = 2..1 OSI) - corespunde partii inferioare a subsistemului retea OSI

Protocoale si servicii reprezentative la nivel **acces retea: Ethernet**, **Token Ring**, etc.

Protocoale si servicii reprezentative la nivel **retea** (responsabil de rutarea pachetelor Internet):

- **IP** (***Internet Protocol***) - protocol care ofera servicii Internet **fara conexiune**, asigurand transmiterea **nefiabila** a pachetelor IP intre sisteme terminale, pe baza unor adrese unice, specifice fiecarui nod, numite **adrese Internet** sau **adrese IP**; utilizat de **TCP** si **UDP**;
- **ARP** (*Address Resolution Protocol*) si **RARP** (*Reverse Address Resolution Protocol*) - protocoale care realizeaza **translatia dinamica a adreselor Internet in adrese hard unice** (de exemplu Ethernet) **din reteaua locala, si invers**, pe baza tabelelor de translatare ARP;
- **ICMP** (*Internet Control Message Protocol*) - protocol care furnizeaza mesaje de **eroare si control** pentru IP; etc.

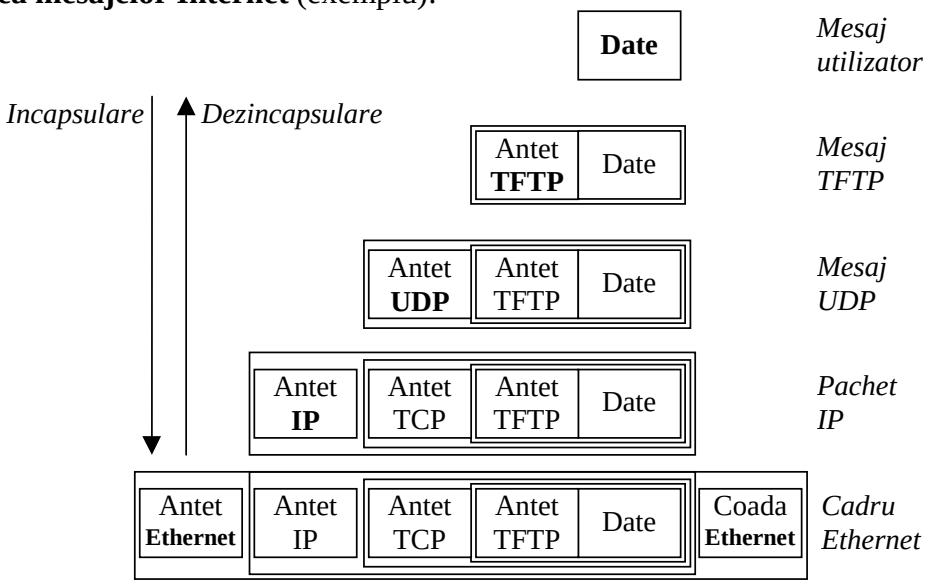
Protocoale si servicii reprezentative la nivel **transport:**

- **TCP** (*Transport Control Protocol*) - protocol care ofera servicii Internet **orientate spre conexiune** (circuite virtuale, similare sistemului **telefonic** clasic), asigurand:
  - transferul **fiabil** (sigur, fara pierderi de informatie, ordonat) al datelor intre aplicatiile sursa si destinatie,
  - in **flux continuu**,
  - **controlul fluxului** de date,
  - **multiplexarea fluxurilor** de date al mai multor procese si **controlul conexiunilor**,
- **UDP** (*User Datagram Protocol*) - protocol care ofera servicii Internet **fara conexiune**, bazate pe **transmisia independenta a fiecarui mesaj** (datagrama UDP, similara unei scrisori), asigurand:
  - transferul **nefiabil** (nesigur, cu pierderi, neordonat),
  - adaugarea unei **sume de control** la pachetele IP, si
  - **multiplexarea** fluxurilor de mesaje al mai multor procese; etc.

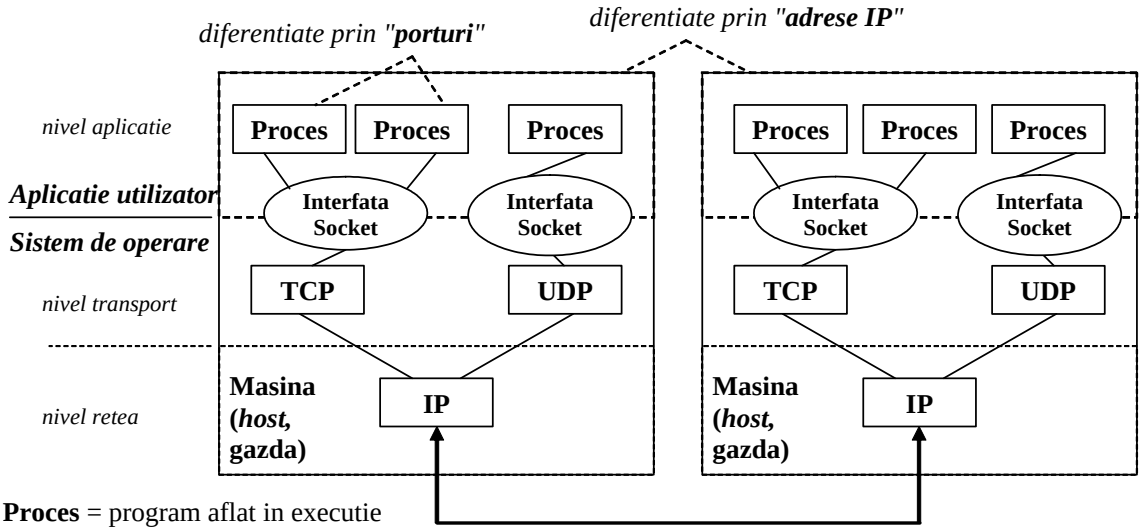
Protocoale si servicii reprezentative la nivel **aplicatie:**

- **FTP** (*File Transfer Protocol*) - protocol pentru transport de fisiere; utilizeaza **TCP**;
- **TFTP** (*Trivial File Transfer Protocol*) - protocol pentru transport de fisiere; utilizeaza **UDP**;
- **HTTP** (*Hyper-Text Transfer Protocol*) - protocol pentru transport de *hyper-pagini*; utilizeaza **TCP**;
- **Telnet** - protocol pentru transferul la distanta (pe un alt sistem terminal) al actiunilor utilizatorului; utilizeaza **TCP**; etc.
- **NFS** (*Network File System*) - protocol pentru accesul in comun al sistemelor de fisiere al terminalelor dintr-o retea locala de catre utilizatori, ca unic sistem de fisiere; utilizeaza **TCP**.

Incapsularea mesajelor Internet (exemplu):



5.1.5. Detalii utile in programare privind protocoalele Internet



Adresele IP au **32 biti** (32b), adica **4 octeti** (4B). Pentru reprezentare se folosesc diferite forme:

- **numar binar** (reprezentarea corespunzatoare stocarii in sistemele de calcul, dar cea mai ineficienta, si neutilizata de catre programator); exemplu:
  10001111 01010101 00101011 00001111
- **numar zecimal** (reprezentare neutilizata de programatorilor, desi e forma uzuala de reprezentare matematica); exemplu: 2 404 723 471
- **numar hexazecimal** (reprezentare condensata a numarului binar, neutilizata de programator); exemplu: 0x 8F 55 2B 0F
- **notatie zecimala cu puncte** ("*dotted quad*" sau "*dotted decimal*", reprezentare utilizata de programator); exemplu: 143**.**85**.**43**.**15

Clase de adrese IP:

Clasa adrese	Primii biti	Lungime ID retea (adresa retea)	Lungime ID host	Exemplu	Observatii
A	**0**	**1 B = 8 b** valori **0** .. **126**	**3 B = 32 b**	124.95.44.15	Alocate retelelor de dimensiuni **mari**, deoarece fiecare ID retea corespunde unui numar de **peste 16 milioane** ID-uri host.
B	**1 0**	**2 B = 16 b** valori **128**.000 .. **191**.255	**2 B = 16 b**	151.10.13.28	Alocate retelelor de dimensiuni **medii**, deoarece fiecare ID retea corespunde unui numar de **peste 65 000** ID-uri host
C	1 **1 0**	**3 B = 32 b** valori **192**.000.000 .. **223**.255.255	**1 B = 8 b**	201.110.213.2	Alocate retelelor de dimensiuni **mici**, deoarece fiecare ID retea corespunde unui numar de **254** ID-uri host
D, E	1 1 **1**	rezervate			

Adresele IP ale caror componente (adrese sau ID-uri) host au toti bitii egali cu 0 sunt rezervate pentru **adresarea retelei**. De exemplu, adresa de clasa A "**110**.0.0.0" contine *host-ul* "110.23.52.12". Un exemplu similar pentru clasa B poate fi "**186**.**10**.0.0" pentru reteaua cu adresa "**186**.**10**".

O facilitate asociata adreselor de retea este **difuzarea** (***broadcast-ul***) pachetelor IP catre intreaga retea de clasa A "**110**", prin trimiterea lor pe adresa "**110**.255.255.255", respectiv *broadcast-ul* pachetelor IP catre intreaga retea de clasa B "**186.10**", prin trimiterea lor pe adresa "**186.10**.255.255".

Porturile TCP si UDP sunt identificate prin numere de port de **16 biti** (16b), adica **2 octeti** (2B). Alocarea numerelor de port standard (rezervate):

- 0 .. 255 - alocate unor **protocoale de nivel aplicatie publice, standard**; de exemplu:
  - 7 - protocol ecou (*echo*)
  - 21 - FTP
  - 23 - Telnet
  - 25 - SMTP (Simple Mail Transfer Protocol)
  - 69 - TFTP
  - 80 - HTTP
- 256 .. 1023 - alocate unor **companii** de pe piata comunicatiilor;
- 1024 .. 65 535 - **nealocate**, libere pentru aplicatii Internet.

Pentru comunicatia intre doua procese care se desfasoara pe doua masini **este necesar sa se specifice**:

- adresele celor doua retele in care se afla host-urile;
- adresele celor doua host-uri in cadrul retelor lor;
- porturile la care procesele sunt conectate (prin intermediul sistemul de operare) la host-uri;
- protocolul utilizat.

Legatura dintre procese aflate pe masini (gazde, *host-uri*) diferite poarta numele de **conexiune** (*connection*). Perechea **{adresa retea, adresa host}** formeaza **adresa IP** a masinii. Specificarea unei comunicatii intre doua procese aflate pe masini diferite se poate reduce la un cvintuplu, cunoscut sub numele de **asociere** (*association*):

{ Protocol , Adresa IP sursa , Port proces sursa , Adresa IP destinatie , Port proces destinatie}

De exemplu, **pentru o conexiune TCP putem specifica asocierea**:

{ **TCP** , **143.85.43.26** , **3000** , **143.85.43.37** , **4000** }

Pentru o conexiune se pot defini doua **semiasocieri** (*half associations*):

{ Protocol , Adresa IP sursa , Port proces sursa }
{ Protocol , Adresa IP destinatie , Port proces destinatie }

A doua semiasociere poate fi folosita pentru **specificarea destinatiei unor datagrame UDP.** De exemplu:

{ **UDP** , **143.85.43.37** , **4000** }

Ca alternativa la adresele numerice IP exista **numele date host-urilor IP** (sub forma de siruri de caractere), mult mai naturala pentru utilizatori. De exemplu, astfel de nume de host poate fi "**host1.elcom.pub.ro**" unde sectiunea "**elcom.pub.ro**" corespunde domeniului asociat retelei locale, iar sectiunea "**host1**" specifica host-ul in cadrul retelei locale. Adresele complete ale utilizatorilor (de e-mail de exemplu) provin din acestea (de exemplu "user1@host1.elcom.pub.ro").

Asocierile adreselor numerice IP cu numele date host-urilor IP se face la nivelul sistemului de operare, si pot fi obtinute de catre utilizator in mai multe moduri (care fac apel la **serverele domeniilor de nume - DNS** - din Internet).

De exemplu, sub Unix dar si alte sisteme de operare, serviciul **nslookup** permite interogarea DNS pentru obtinerea atat a adresei in forma numerica (*dotted quad*) cat si a numelui de host, atunci cand este furnizata fie adresa in forma numerica fie numele host-ului. Prin apelul comenzii **nslookup** cu parametru "**host1.elcom.pub.ro**" se poate obtine ca raspuns o pereche de genul "**host1.elcom.pub.ro**", "**143.85.41.121**". In reteaua locala (in acest exemplu corespunzatoare domeniului "**elcom.pub.ro**"), se poate obtine acelasi rezultat apeland **nslookup** cu parametru "**host1**". De asemenea, apeland **nslookup** cu parametru "**143.85.41.12**" se poate obtine ca raspuns o pereche de genul "**host1.elcom.pub.ro**", "**143.85.41.121**".

5.1.6. Introducere in *socket-uri*

Socket-ul este un **punct final al unei comunicatii intre procese**, care ofera un punct de acces la servicii de nivel transport (TCP sau UDP) in Internet. Pe de alta parte, *socket-ul* este o resursa alocata de sistemul de operare, mentinuta de sistemul de operare, si accesibila utilizatorilor prin intermediul unui intreg numit descriptor de (fisier) *socket*.

Intr-o conexiune intre procese, ***socket-ul*** corespunde unei semiasocieri.

Tipurile de socket oferite de sistemele de operare actuale corespund celor doua clase mari de servicii de comunicatie:

- ***socket-uri* flux** (*stream*) - pentru **servicii orientate spre conexiune** (**TCP**) si
- ***socket-uri* datagrama** (*datagram*) - pentru **servicii fara conexiune** (**UDP**).

***Socket-urile* Berkeley** (create la Universitatea cu acelasi nume din California) sunt primele protocoale punct–la–punct pentru comunicatii pe baza de stiva TCP/IP. *Socket-urile* au fost introduse in 1981, ca **interfata generica** Unix BSD 4.2 **care suporta comunicatii interproces** (IPC = *interprocess communications*) **Unix–la–Unix**.

Astazi, *socket-urile* sunt suportate de orice sistem de operare. API-ul *Windows* pentru *socket-uri*, cunoscut ca *WinSock* (mai nou *WinSock2*)este o specificatie care standardizeaza folosirea TCP/IP sub sistemul de operare *Windows*. *In sistemele Unix BSD*, *socket-urile* fac parte din nucleu; ele ofera si servicii de sine statoare si de comunicatii intre procese. *Sistemele non – Unix BSD*, MS – DOS, Windows, MacOS, and OS/2 ofera *socket-urile* sub forma unor biblioteci.

Java ofera *socket-urile* ca parte a unei biblioteci de clase standard, `java.net`.

Practic, *socket-urile* ofera standardele portabile curente *de facto* pentru furnizorii de aplicatii pentru retele pe retelele TCP/IP.

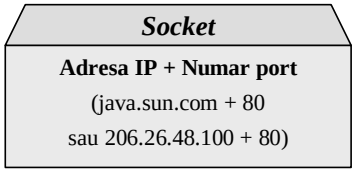
***Socket-urile* datagrame** ofera o interfata la protocolul Internet de transport prin datagrame **UDP** (*User Datagram Protocol*). UDP se ocupa de transmisile prin retea sub forma de **pachete independente** (datagrame) si nu ofera garantii privind vreunul dintre parametrii QoS (rata, intarziere, jitter, pierderi, etc.). UDP nu include verificari de sume, nu incearca sa detecteze pachetele duplicate sau sa mentina orice forma de secventialitate a transmisiunilor multipachet. Protocolul nu prevede nici o confirmare a primirii pachetelor. Informatiile pot fi pierdute, duplicate sau primite in ordine gresita. Aplicatia care utilizeaza UDP-ul este raspunzatoare de retransmisii, cand se petrece o eroare. Un avantaj este ca datagramele se deplaseaza repede si introduc un *overhead* mic. Un dezavantaj este ca datagramele impun o limita a datelor transferate (tipic 64 KB), si necesita ca aplicatia sa-i ofere toate mecanismele de asigurare a fiabilitatii.

***Socket-urile* flux** ofera o interfata la protocolul Internet de transport orientat spre conexiune **TCP** (*Transport Control Protocol*). TCP ofera un serviciu bazat pe sesiune care include controlul fluxului, reasamblarea pachetelor si mentinerea conexiunii. Socket-urile flux garanteaza ca pachetele se trimit fara erori sau duplicari si ca sunt primite in aceeasi ordine in care s-au transmis. Nu este impusa nici o limita informatiei; se considera a fi un **flux de biti**. Ca avantaj, TCP ofera un mecanism fiabil punct–la–punct. Ca dezavantaj, TCP este mai incet decat UDP si necesita mai mult *overhead* de programare.

5.2. Incapsularea adreselor IP in limbajul Java

5.2.1. Incapsularea adreselor IP

O **adresa socket** pe o retea bazata pe IP consta din doua parti: o **adresa IP** si o adresa de port (**numar de port**):



Socket-ul = Adresa IP + Numarul de port

O adresa de Internet (adresa IP) este un numar de **32 biti** (4 octeti), de obicei reprezentat ca un sir de 4 valori numerice intre 0 si 255 despartite prin puncte (de exemplu **206.26.48.100**). Adresa IP mai poate fi reprezentata prin numele domeniului (de exemplu **java.sun.com**).

Un port este un **punct de intrare (dinspre retea) intr-o aplicatie** care se afla pe o masina gazda. El este reprezentat de un numar pe **16 biti** (2 octeti).

5.2.2. Clasa java.net.InetAddress – interfata publica

Pachetul **`java.net`** foloseste clasa **`InetAddress`** pentru a incapsula o adresa IP. In continuare sunt **detaliate cateva dintre metodele** declarate in clasa **`InetAddress`**:

`boolean`	**`equals`**`(Object obj)` Compara obiectul curent (`this`) cu obiectul primit ca parametru (`obj`).
`byte[]`	**`getAddress`**`()` Returneaza adresa IP incapsulata in obiectul caruia i se aplica, sub forma numerica (*raw IP address*). Cei 4 octeti returnati sunt in ordine NBO (*network byte order*), adica octetul de ordin maxim al adresei poate fi gasit in `getAddress()[0]`.
`static InetAddress`	**`getByAddress`**`(byte[] addr)` Returneaza obiectul `InetAddress` care incapsuleaza adresa IP numerica pasata.
`static InetAddress`	**`getByName`**`(String host)` Returneaza obiectul `InetAddress` care incapsuleaza adresa IP a gazdei a carui nume i-a fost pasat (numele gazdei poate fi un nume de masina, de exemplu **`java.sun.com`**, sau adresa IP numerica).
`String`	**`getHostAddress`**`()` Returneaza adresa IP incapsulata in obiectul caruia i se aplica, sub forma de sir de caractere (obiect `String`).
`String`	**`getHostName`**`()` Returneaza numele gazdei a carei adresa IP este incapsulata in obiectul curent.
`static InetAddress`	**`getLocalHost`**`()` Returneaza obiectul `InetAddress` care incapsuleaza adresa IP locala.
`String`	**`toString`**`()` Converteste adresa IP curenta la `String`.

Clasa **`InetAddress` incapsuleaza o adresa IP** intr-un obiect care **poate intoarce informatia utila**. Aceasta informatie utila se obtine invocand metodele unui obiect al acestei clase. De exemplu, metoda **`equals()`** intoarce adevarat daca doua obiecte reprezinta aceeasi adresa IP.

Clasa `InetAddress` nu are constructor public. De aceea, **pentru a crea obiecte ale acestei clase trebuie invocata una dintre metodele** de clasa **`getLocalHost()`** si **`getByName()`**.

Codul:

```
byte[] octetiAdresaServer = { 200, 26, 48, 100 };
InetAddress adresaServer = InetAddress.getByAddress(octetiAdresaServer);
```

este echivalent cu:

```
String numeMasinaServer = "java.sun.com";
InetAddress adresaServer = InetAddress.getByName(numeMasinaServer);
```

si cu:

```
String adresaIPMasinaServer = "200.26.48.100";
InetAddress adresaServer = InetAddress.getByName(adresaIPMasinaServer);
```

Pentru a obtine **obiectul `InetAddress` care incapsuleaza adresa IP locala** poate fi folosit apelul:

```
InetAddress.getLocalHost()
```

Urmatorul program **afiseaza informatii privind masina locala**.

```
 1  import java.net.*;
 2
 3  class InfoLocalHost {
 4    public static void main (String args[]) {
 5      try {
 6
 7        InetAddress adresaLocala = InetAddress.getLocalHost();
 8
 9        System.out.println("Numele meu este " +
10                                 adresaLocala.getHostName());
11
12        System.out.println("Adresa mea IP este " +
13                                 adresaLocala.getHostAddress());
14
15        System.out.println("Octetii adresei mele IP sunt {" +
16            adresaLocala.getAddress()[0] + ", " +
17            adresaLocala.getAddress()[1] + ", " +
18            adresaLocala.getAddress()[2] + ", " +
19            adresaLocala.getAddress()[3] + "}");
20      }
21      catch (UnknownHostException e) {
22        System.out.println("Imi pare rau. Nu imi cunosc numele.");
23      }
24    }
25  }
```

O **adresa IP speciala** este **adresa IP *loopback*** (tot ce este trimis catre aceasta adresa IP se intoarce si devine intrare IP pentru gazda locala), **cu ajutorul careia pot fi testate local programe care utilizeaza *socket-uri***.

Numele "**`localhost`**" si valoarea numerica "**`127.0.0.1`**" sunt folosite pentru a identifica adresa IP *loopback*.

Pentru a obtine **obiectul `InetAddress` care incapsuleaza adresa IP *loopback*** pot fi folosite apelurile:

```
InetAddress.getByName(null)
InetAddress.getByName("localhost")
InetAddress.getByName("127.0.0.1")
```

Metoda `getAddress()` returneaza octetii adresei IP incapsulate, ceea ce poate fi util pentru filtrarea adreselor.

Urmatorul program permite **obtinerea si afisarea informatiilor privind o masina specificata**, inclusiv **clasa adresei IP**.

```
 1  import java.net.*;
 2  import java.io.*;
 3  import javax.swing.*;
 4
 5  public class InformatiiMasinaIP {
 6    public static void main(String args[]) {
 7      String nume;
 8      do {
 9        nume = JOptionPane.showInputDialog
10                    ("Introduceti un nume de masina sau o adresa IP");
11        if (nume != null)
12          afisareaInformatiilor(nume);
13      } while (nume != null);
14    }
15
16    static char clasaIP(byte[] octetiAdresa) {
17      int octetSuperior = 0xff & octetiAdresa[0];
18      return (octetSuperior < 128) ? 'A' : (octetSuperior < 192) ? 'B' :
19        (octetSuperior < 224) ? 'C' : (octetSuperior < 240) ? 'D' : 'E';
20    }
21
22    static void afisareaInformatiilor(String nume) {
23      try {
24        InetAddress masina = InetAddress.getByName (nume);
25        System.out.println ("Numele masinii: " + masina.getHostName());
26        System.out.println ("IP-ul masinii: " + masina.getHostAddress());
27        System.out.println ("Clasa masinii: " +
28                                      clasaIP(masina.getAddress()));
29      } catch (UnknownHostException ex) {
30        System.out.println("Nu am reusit sa gasesc " + nume);
31      }
32    }
33  }
```

Programul urmator contine o metoda de clasa (globala, utilitara) **`afisareInetAddress()`**, pentru **obtinerea si afisarea informatiilor privind o masina specificata**.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5   *  Afisare informatii privind adresa IP
 6   */
 7  public class InfoInetAddress {
 8
 9    /**
10     *  Afiseaza informatii privind adresa IP specificata ca parametru
11     */
12    public static void afisareInetAddress(InetAddress ia, String nume) {
13      System.out.println("\n -------------------------------------");
14      System.out.println("\n Informatii privind adresa IP " + nume);
15      System.out.println();
16
17      System.out.println("\t Adresa IP ca text: " +
18              ia.getHostAddress());
19
20      System.out.println("\t Numele masinii: " +
21              ia.getHostName());
22
23      System.out.println("\t Numele calificat: " +
24              ia.getCanonicalHostName());
25
26      System.out.println("\t Octetii adresei IP: " +
27              ia.getAddress()[0] + " " + ia.getAddress()[1] + " " +
28              ia.getAddress()[2] + " " + ia.getAddress()[3]);
29
30      System.out.println("\t Obiectul InetAddress convertit la String: "+
31              ia.toString());
32    }
33
34    /**
35     *  Metoda principala cu rol de test
36     *  si exemplificare a modului de utilizare
37     */
38    public static void main (String args[]) {
39
40      InetAddress adresaLocala;
41
42      try {
43        adresaLocala = InetAddress.getLocalHost();
44      }
45      catch (UnknownHostException ex) {
46        System.err.println("Nu poate fi obtinuta adresa locala");
47      }
48
49      afisareInetAddress(adresaLocala, "adresa locala");
50
51      afisareInetAddress(adresaLocala, adresaLocala.toString());
52
53      afisareInetAddress(adresaLocala, null);
54
55      afisareInetAddress(null, null);
56    }
57  }
```

Metoda **`main()`** a programului ilustreaza modul de lucru cu metoda **`afisareInetAddress()`** pentru afisarea informatiilor privind masina locala.

5.3. Socket-uri flux (TCP)

5.3.1. Lucrul cu socket-uri flux (TCP)

Java ofera, in pachetul **`java.net`**, mai multe clase pentru lucrul cu *socket-uri* flux (TCP). Urmatoarele **clase Java** sunt **implicate in realizarea conexiunilor TCP** obisnuite: `ServerSocket` si `Socket`.

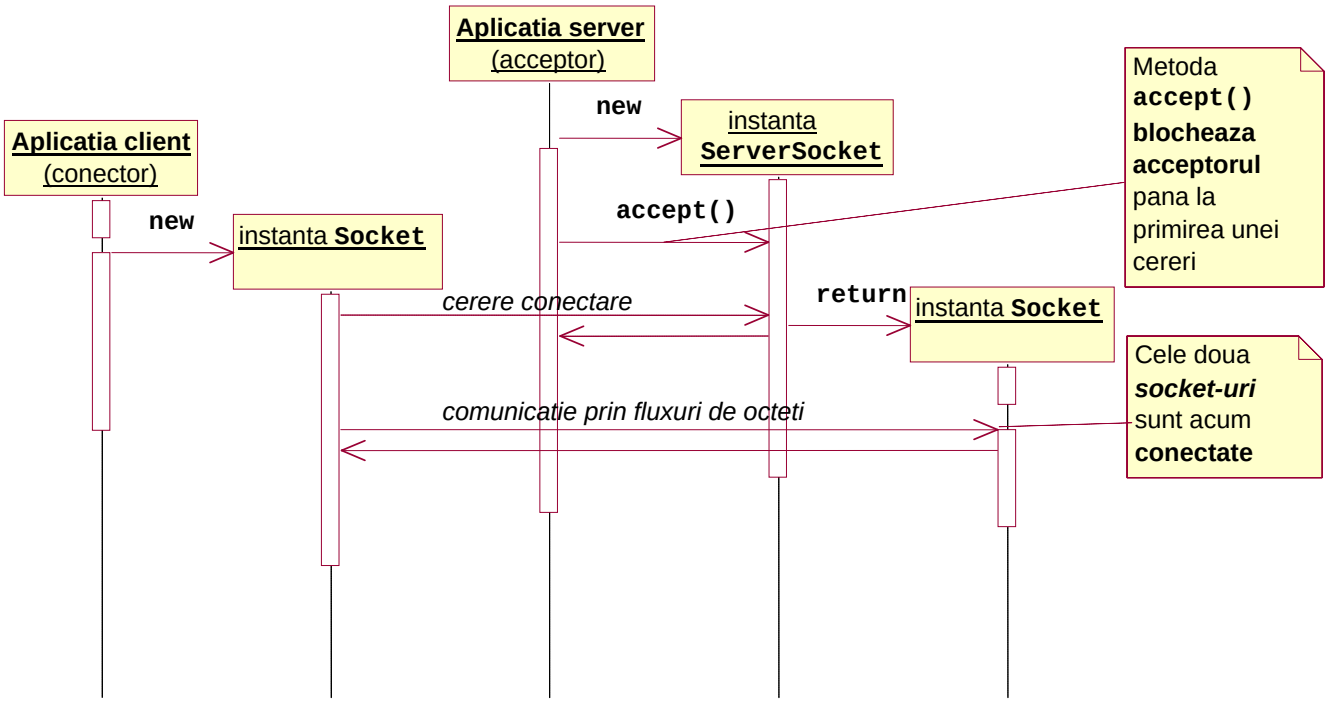
Clasa **`ServerSocket`** reprezinta ***socket-ul*** (aflat eventual pe un server bazat pe TCP) **care asteapta si accepta cereri de conexiune** (eventual de la un client bazat pe TCP).

Clasa **`Socket`** reprezinta **punctul terminal al unei conexiuni TCP** intre doua masini (eventual un client si un server).

Clientul (sau, mai general, **masina conector**) **creeaza un punct terminal `Socket`** in momentul in care cererea sa de conexiune este lansata si acceptata.

Serverul (sau, mai general, **masina acceptor**) **creeaza un `Socket`** in momentul in care primeste si accepta o cerere de conexiune, **si continua sa asculte si sa astepte alte cereri pe `ServerSocket`**.

Secventa tipica a mesajelor schimbate intre client si server este urmatoarea:



Odata conexiunea stabilita, metodele `getInputStream()` si **`getOutputSteam()`** ale clasei **`Socket`** trebuie **utilizate pentru a obtine fluxuri de octeti**, de intrare respectiv iesire, pentru comunicatia intre aplicatii.

5.3.2. Clasa socket flux (TCP) pentru conexiuni (Socket) – interfata publica

1. Principalii constructori ai clasei `Socket`:

`Socket``()` Creaza un socket flux neconectat, cu implementarea (`SocketImpl`) implicita platformei.
`Socket``(String host, int port)` Creaza un socket flux si il conecteaza la portul de numar specificat, la masina a carui nume este specificat.
`Socket``(InetAddress address, int port)` Creaza un socket flux si il conecteaza la portul de numar specificat, la adresa IP specificata.
`Socket``(String host, int port, InetAddress localAddr, int localPort)` Creaza un socket flux si il conecteaza la portul de numar specificat, la masina a carui nume este specificat. Socket-ul se va si lega (*bind*) la portul local specificat, la adresa IP locala specificata.
`Socket``(InetAddress address, int port, InetAddress localAddr, int localPort)` Creaza un socket flux si il conecteaza la portul de numar specificat, la adresa IP specificata. Socket-ul se va si lega (*bind*) la portul local specificat, la adresa IP locala specificata.

2. Declaratiile si descrierea catorva metode ale clasei **`Socket`**:

`void`	**`close`**`()` Inchide *socket-ul* curent.
`InetAddress`	**`getInetAddress`**`()` Returneaza un obiect care incapsuleaza adresa IP la care este conectat *socket-ul* curent.
`InputStream`	**`getInputStream`**`()` Returneaza un flux de intrare a octetilor dinspre *socket-ul* curent.
`boolean`	**`getKeepAlive`**`()` Testeaza (returneaza true daca este validata) optiunea SO_KEEPALIVE.
`InetAddress`	**`getLocalAddress`**`()` Returneaza un obiect care incapsuleaza adresa IP locala la care *socket-ul* curent este legat.
`int`	**`getLocalPort`**`()` Returneaza numarul portului local la care *socket-ul* curent este legat.
`OutputStream`	**`getOutputStream`**`()` Returneaza un flux de iesire a octetilor catre socket-ul curent.
`int`	**`getPort`**`()` Returneaza numarul portului la care *socket-ul* curent este conectat.
`int`	**`getReceiveBufferSize`**`()` Returneaza valoarea optiunii SO_RCVBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de intrare dinspre *socket*.
`int`	**`getSendBufferSize`**`()` Returneaza valoarea optiunii SO_SNDBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de iesire catre *socket*.
`boolean`	**`getTcpNoDelay`**`()` Testeaza (returneaza true daca este validata) optiunea SO_KEEPALIVE (algoritmul Nagle, de confirmare pozitiva, este activat).

`int`	**`getSoTimeout`**`()` Returneaza valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`read()`** asupra fluxului de intrare asociat *socket-ului* curent blocheaza aplicatia, asteptand sosirea unor octeti prin flux. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`int`	**`getTrafficClass`**`()` Obtine clasa de trafic (*traffic class*) sau tipul de serviciu (ToS) din antetul IP al pachetelor trimise prin *socket-ul* curent.
`boolean`	**`isClosed`**`()` Returneaza true daca *socket-ul* curent este inchis, altfel returneaza false.
`boolean`	**`isConnected`**`()` Returneaza true daca *socket-ul* curent este conectat cu succes, altfel returneaza false.
`boolean`	**`isInputShutdown`**`()` Returneaza true daca fluxul de intrare al *socket-ului* curent este "la sfarsit de flux" (EOF), altfel returneaza false.
`boolean`	**`isOutputShutdown`**`()` Returneaza true daca fluxul de iesire al *socket-ului* curent este valid, altfel returneaza false.
`void`	**`setKeepAlive`**`(boolean on)` Valideaza/invalideaza optiunea SO_KEEPALIVE.
`void`	**`setReceiveBufferSize`**`(int size)` Stabileste valoarea optiunii SO_RCVBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de intrare dinspre *socket*.
`void`	**`setSendBufferSize`**`(int size)` Stabileste valoarea optiunii SO_SNDBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de iesire catre *socket*.
`void`	**`setSoTimeout`**`(int timeout)` Stabileste valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`read()`** blocheaza aplicatia, asteptand sosirea unor octeti prin fluxul de intrare. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`void`	**`setTcpNoDelay`**`(boolean on)` Valideaza/invalideaza optiunea TCP_NODELAY (Activeaza/inactiveaza algoritmul Nagle).
`void`	**`setTrafficClass`**`(int tc)` Stabileste clasa de trafic (*traffic class*) sau tipul de serviciu (ToS) din antetul IP al pachetelor trimise prin *socket-ul* curent.
`void`	**`shutdownInput`**`()` Plaseaza fluxul de intrare al *socket-ului* curent "la sfarsit de flux" (EOF).
`void`	**`shutdownOutput`**`()` Invalideaza fluxul de iesire al *socket-ului* curent (implicit este valid).
`String`	**`toString`**`()` Returneaza un `String` continand informatii privind *socket-ul* curent.

In continuare este prezentata **secventa tipica pentru crearea *socket-ului* unei aplicatii conector (client)**:

```
// Stabilirea adresei serverului
String adresaServer = "localhost";

// Stabilirea portului serverului
int portServer = 2000;

// Crearea socketului (implicit este realizata conexiunea cu serverul)
Socket socketTCPClient = new Socket(adresaServer, portServer);
```

ca si pentru **crearea fluxurilor de octeti asociate *socket-ului***:

```
// Obtinerea fluxului de intrare octeti TCP
InputStream inTCP = socketTCPClient.getInputStream();

// Obtinerea fluxului de intrare caractere dinspre retea
InputStreamReader inTCPCaractere = new InputStreamReader(inTCP);
// Adaugarea facilitatilor de stocare temporara
BufferedReader inRetea = new BufferedReader(inTCPCaractere);

// Obtinerea fluxului de iesire octeti TCP
OutputStream outTCP = socketTCPClient.getOutputStream();

// Obtinerea fluxului de iesire spre retea,
// cu facilitate de afisare (similare consolei standard de iesire)
PrintStream outRetea  = new PrintStream(outTCP);
```

Socket-ul poate fi utilizat pentru **trimiterea de date**:

```
// Crearea unui mesaj
String mesajDeTrimis = "Continut mesaj";

// Scrierea catre retea (trimiterea mesajului)
outRetea.println(mesajDeTrimis);

// Fortarea trimiterii
outRetea.flush();
```

sau pentru **primirea de date**:

```
// Citirea dinspre retea (receptia unui mesaj)
String mesajPrimit = inRetea.readLine();

// Afisarea mesajului primit
System.out.println(mesajPrimit);
```

Dupa utilizare, *socket-ul* poate fi **inchis**:

```
// Inchiderea socketului (implicit a fluxurilor TCP)
socketTCPClient.close();
```

Metoda **`setTrafficClass`**(`int` **`tc`**) stabileste clasa de trafic (*traffic class*) sau tipul de serviciu (ToS) din antetul IP al pachetelor trimise prin *socket-ul* curent. **Deoarece implementarea nivelului retea poate ignora valoarea parametrului `tc`** (in cazul in care **nu exista sau nu este activat controlul de trafic pentru servicii diferentiate**), **aplicatia trebuie sa interpreteze apelul ca pe o sugestie** (*hint*) data nivelului inferior.

Pentru protocolul IPv4 valoarea parametrului **`tc`** este interpretata drept **campurile precedenta si ToS**. **Campul ToS este creat ca set de biti** obtinut prin **aplicarea functiei logice OR valorilor**:

`IPTOS_LOWCOST = 0x02` - indicand cerinte de **cost redus** din partea aplicatiei
`IPTOS_RELIABILITY = 0x04` - indicand cerinte de **fiabilitate** din partea aplicatiei
`IPTOS_THROUGHPUT = 0x08` - indicand cerinte de **banda larga** din partea aplicatiei
`IPTOS_LOWDELAY = 0x10` - indicand cerinte de intarziere redusa (**timp real**) ale aplicatiei

Cel mai putin semnificativ bit al octetului **`tc`** e ignorat, el **trebuind sa fie zero** (*must be zero* – MZB). **Stabilirea bitilor in campul de precedenta** (cei mai semnificativi 3 biti ai octetului **`tc`**) poate produce o **`SocketException`**, indicand imposibilitatea realizarii operatiei.

Pentru protocolul IPv6 valoarea parametrului **`tc`** este plasata in **campul `sin6_flowinfo`** al antetului IP.

In continuare este prezentata o **aplicatie conector (client)**, care permite **obtinerea si afisarea informatiilor privind *socket-ul* creat**. Programul face apel la metoda globala, utilitara, `afisareInetAddress()`, a clasei `InfoInetAddress` (anterior creata) pentru obtinerea si afisarea informatiilor privind o masina specificata.

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Afisare informatii privind conexiunea TCP
 5   */
 6  public class InfoSocketTCP {
 7  /**
 8   *  Afiseaza informatii privind conexiunea TCP specificata ca parametru
 9   */
10    public static void afisareInfoSocketTCP (Socket socketTCP) {
11      System.out.println("\nConexiune de la adresa " + socketTCP.getLocalAddress() +
12                         " de pe portul " + socketTCP.getLocalPort() +
13                         " catre adresa " + socketTCP.getInetAddress() +
14                         " pe portul " + socketTCP.getPort() + "\n");
15    }
16    /**
17     *  Metoda principala cu rol de test si exemplificare a modului de utilizare
18     */
19    public static void main (String args[]) {
20      BufferedReader inConsola  = new BufferedReader(new
21                                      InputStreamReader(System.in));
22      String adresaIP = null;
23      int numarPort = 0;
24      Socket conexiuneTCP;
25      try {
26        System.out.print("Introduceti adresa IP dorita: ");
27        adresaIP = inConsola.readLine();
28
29        System.out.print("Introduceti numarul de port dorit: ");
30        numarPort = Integer.parseInt(inConsola.readLine());
31
32        conexiuneTCP = new Socket(adresaIP, numarPort);
33
34        afisareInfoSocketTCP(conexiuneTCP);
35
36        InetAddress iaServer = conexiuneTCP.getInetAddress();
37        InfoInetAddress.afisareInetAddress(iaServer, "adresa serverului");
38
39        InetAddress iaLocala = conexiuneTCP.getLocalAddress();
40        InfoInetAddress.afisareInetAddress(iaLocala, "adresa locala");
41
42        conexiuneTCP.close();
43      }
44      catch (NumberFormatException ex) {
45        System.err.println("Numarul de port nu are format intreg");
46      }
47      catch (UnknownHostException ex) {
48        System.err.println("Nu poate fi gasita " + adresaIP);
49      }
50      catch (SocketException ex) {
51        System.err.println("Nu se poate face conexiune cu " + adresaIP +
52                                                          ":" + numarPort);
53      }
54      catch (IOException ex) {
55        System.err.println(ex);
56      }
57    }
58  }
```

5.3.3. Clasa socket TCP pentru server (ServerSocket) – interfata publica

1. Principalii constructori ai clasei **`ServerSocket:`**

`ServerSocket``()` Creaza un socket pentru server (de tip acceptor) nelegat la vreun port.
`ServerSocket``(int port)` Creaza un *socket* pentru server (de tip acceptor) legat la portul local de numar specificat. Valoarea 0 va conduce la crearea unui *socket* legat la un port liber nespecificat explicit. Numarul de indicatii privind cererile de conexiune care pot sta in asteptare la un moment dat este implicit 50.
`ServerSocket``(int port, int backlog)` Creaza un *socket* pentru server (de tip acceptor) legat la portul local de numar specificat. Valoarea 0 va conduce la crearea unui *socket* legat la un port liber nespecificat explicit. Numarul de indicatii privind cererile de conexiune care pot sta in asteptare la un moment dat este specificat de `backlog`.
`ServerSocket``(int port, int backlog, InetAddress bindAddr)` Creaza un *socket* pentru server (de tip acceptor) legat la portul local de numar specificat, la adresa locala specificata. Valoarea 0 va conduce la crearea unui *socket* legat la un port liber nespecificat explicit. Numarul de indicatii privind cererile de conexiune care pot sta in asteptare la un moment dat este specificat de `backlog`.

2. Declaratiile si descrierea catorva metode ale clasei **`ServerSocket`**:

`Socket`	**`accept`**`()` Asteapta cereri de conexiune facute catre *socket-ul* curent si le accepta. Metoda blocheaza executia pana cand e primita o cerere de conexiune. Metoda returneaza un obiect `Socket` prin care se poate desfasura comunicatia utilizand fluxuri de octeti.
`void`	**`close`**`()` Inchide *socket-ul* curent.
`InetAddress`	**`getInetAddress`**`()` Returneaza adresa IP locala a *socket-ului* curent.
`int`	**`getLocalPort`**`()` Returneaza numarul de port local pe care asculta *socket-ul* curent.
`int`	**`getReceiveBufferSize`**`()` Returneaza valoarea optiunii SO_RCVBUF pentru *`ServerSocket`-ul* curent, adica dimensiunea *buffer-ului* propus a fi utilizat de pentru fluxurile de intrare dinspre *socket-urile* obtinute prin acceptarea conexiunilor prin *socket-ul* curent.
`int`	**`getSoTimeout`**`()` Returneaza valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`accept()`** asupra *socket-ului* curent blocheaza aplicatia, asteptand sosirea unor cereri de conexiune. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`boolean`	**`isClosed`**`()` Returneaza true daca *socket-ul* curent este inchis, altfel returneaza false.
`void`	**`setReceiveBufferSize`**`(int size)` Stabileste valoarea optiunii SO_RCVBUF pentru fluxurile de intrare dinspre *socket-urile* obtinute prin acceptarea conexiunilor prin *socket-ul* curent.

`void`	**`setSoTimeout`**`(int timeout)` Stabileste valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`accept()`** blocheaza aplicatia, asteptand sosirea unor cereri de conexiune. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`String`	**`toString`**`()` Returneaza un `String` continand informatii (adresa IP si numarul de port) privind *socket-ul* curent.

In continuare este prezentata **secventa tipica pentru crearea *socket-ului* server al unei aplicatii acceptor (server)**:

```
// Stabilirea portului serverului
int portServer = 2000;

// Crearea socketului server (care accepta conexiunile)
ServerSocket serverTCP = new ServerSocket(portServer);

System.out.println("Server in asteptare pe portul "+portServer+"...");
```

ca si pentru **crearea *socket-ului* pentru tratarea conexiunii TCP cu un client**:

```
// Blocare in asteptarea cererii de conexiune - in momentul acceptarii
// cererii se creaza socketul care serveste conexiunea
Socket conexiuneTCP = serverTCP.accept();

System.out.println("Conexiune TCP pe portul " + portServer + "...");
```

Un caz special este acela in care se creaza un ***socket* server** pentru o aplicatie acceptor **fara a se preciza portul pe care asculta serverul pentru a primi cereri de conexiune si a le accepta**. In acest caz, un numar de port aleator este alocat, dintre cele neocupate in acel moment. Acest lucru se realizeaza **prin pasarea** valorii **`0`** constructorului **`ServerSocket()`**.

Programul urmator ilustreaza **crearea unui *socket* server cu numar de port alocat aleator**.

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class InfoPortTCPAleator {
 4    public static void main (String args[]) {
 5      try {
 6        ServerSocket serverTCP = new ServerSocket(0);
 7        System.out.println("\nAcest server ruleaza pe portul " +
 8                           serverTCP.getLocalPort());
 9        serverTCP.close();
10      }
11      catch (IOException ex) {
12        System.err.println(ex);
13      }
14    }
15  }
```

Urmatorul program ilustreaza crearea de ***socket-uri* server** cu scopul **scanarii conexiunilor TCP de pe masina locala**. Programul **identifica porturile locale pe care exista conexiuni** (**pe care nu pot fi create servere**).

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class ScannerPorturiTCPLocale {
 4    public static void main (String args[]) {
 5      ServerSocket serverTCP;
 6
 7      for (int portTCP=1; portTCP < 65536; portTCP++) {
 8        try {
 9
10          // Urmatoarele linii vor genera exceptie prinsa de blocul catch
11          // in cazul in care e deja un server pe portul portTCP
12          serverTCP = new ServerSocket(portTCP);
13          serverTCP.close();
14
15        }
16        catch (IOException ex) {
17          System.err.println("Exista un server TCP pe portul " + portTCP);
18        }
19      }
20    }
21  }
```

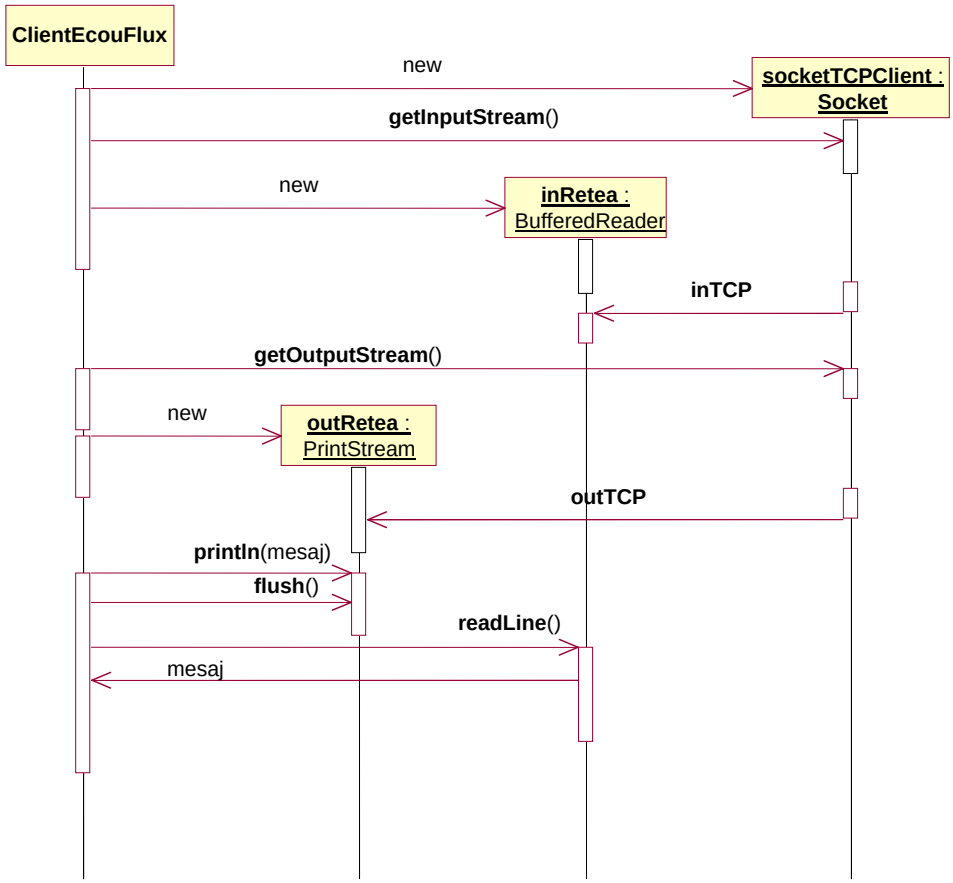
5.3.4. Clienti pentru servere flux (TCP)

Urmatorul program este un **client pentru un server ecou TCP care permite trimiterea unui singur mesaj.**

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  public class ClientEcouFlux {
 5    public static void main(String args[]) throws IOException {
 6      // Stabilirea adresei serverului
 7      String adresaServer = "localhost";
 8
 9      // Stabilirea portului serverului
10      int portServer = 2000;
11
12      // Obtinerea fluxului de intrare caractere dinspre consola
13      InputStreamReader inConsolaCaractere =
14                      new InputStreamReader(System.in);
15
16      // Obtinerea fluxului de intrare caractere dinspre consola,
17      // cu facilitati de stocare temporara
18      BufferedReader inConsola = new BufferedReader(inConsolaCaractere);
19
20      // Obtinerea fluxului de iesire caractere spre consola standard
21      PrintStream outConsola  = System.out;
22
23      // Crearea socketului (conectarea la server)
24      Socket socketTCPClient = new Socket(adresaServer, portServer);
25
26      // Obtinerea fluxului de intrare octeti TCP
27      InputStream inTCP = socketTCPClient.getInputStream();
28
29      // Obtinerea fluxului de intrare caractere dinspre retea
30      InputStreamReader inTCPCaractere = new InputStreamReader(inTCP);
31
32      // Obtinerea fluxului de intrare caractere dinspre retea,
33      // cu facilitati de stocare temporara
34      BufferedReader inRetea = new BufferedReader(inTCPCaractere);
```

```
35
36      // Obtinerea fluxului de iesire octeti TCP
37      OutputStream outTCP = socketTCPClient.getOutputStream();
38
39      // Obtinerea fluxului de iesire caractere, spre retea,
40      // similar consolei standard de iesire
41      PrintStream outRetea  = new PrintStream(outTCP);
42
43
44      // Citirea de la consola a mesajului de trimis
45      outConsola.print("Se trimite: ");
46      String mesajDeTrimis = inConsola.readLine();
47
48      // Scrierea catre retea (trimiterea mesajului)
49      outRetea.println(mesajDeTrimis);
50      // Fortarea trimiterii
51      outRetea.flush();
52
53
54      // Citirea dinspre retea (receptia mesajului)
55      String mesajPrimit = inRetea.readLine();
56
57      // Scrierea la consola (afisarea mesajului)
58      System.out.println("S-a primit: " + mesajPrimit);
59
60      // Inchiderea socketului (implicit a fluxurilor TCP)
61      socketTCPClient.close();
62      System.out.println("Bye!");
63    }
64  }
```

Schimbul de mesaje (creare *socket* si fluxuri, scriere in flux si citire din flux) este ilustrat in urmatoarea diagrama.



Urmatorul program este un **client pentru un server ecou TCP care permite trimiterea mai multor mesaje.** Mesajul format dintr-un punct (".") semnaleaza serverului terminarea mesajelor de trimis, clientul urmand sa isi termine executia.

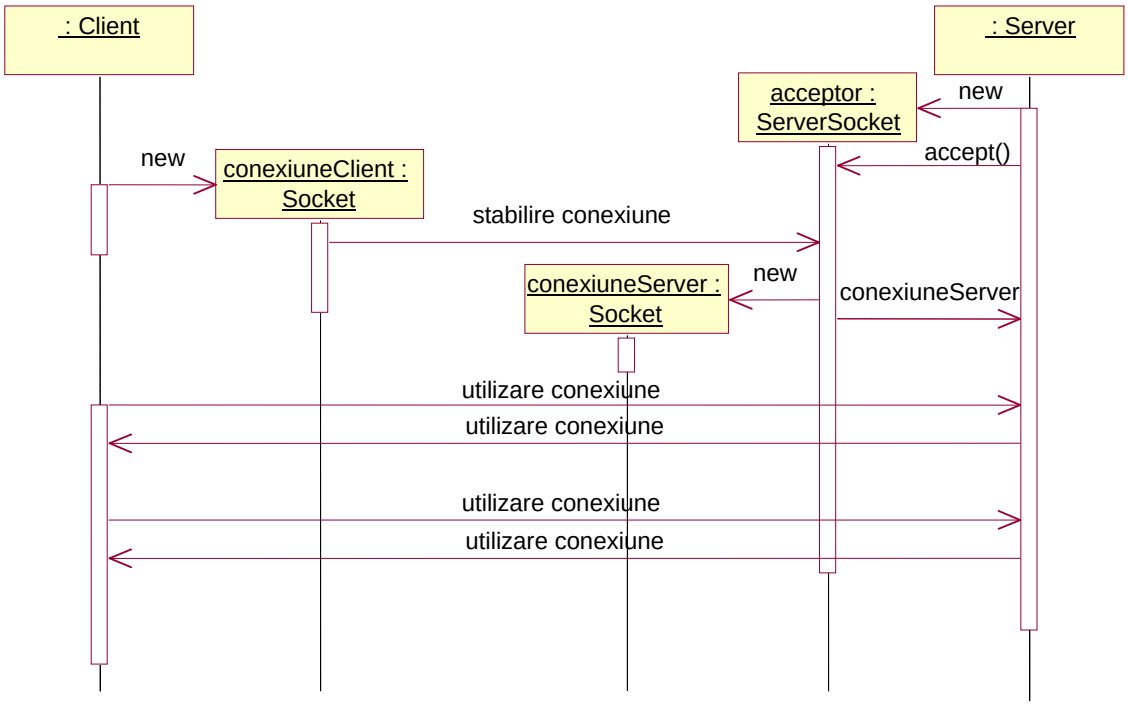
```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5    * Client pentru server ecou flux
 6    */
 7  public class ClientEcouFluxRepetitiv {
 8
 9    public static void main (String args[]) throws IOException {
10      BufferedReader inConsola  = new BufferedReader(new
11                                      InputStreamReader(System.in));
12      System.out.print("Introduceti adresa IP a serverului: ");
13      String adresaServer = inConsola.readLine();
14
15      System.out.print("Introduceti numarul de port al serverului: ");
16      int portServer = Integer.parseInt(inConsola.readLine());
17
18      // Crearea socketului (implicit este realizata conexiunea)
19      Socket conexiuneTCP = new Socket(adresaServer, portServer);
20      System.out.println("Conexiune TCP cu serverul " + adresaServer +
21                         ":" + portServer + "...");
22      System.out.println("Pentru oprire introduceti '.' si <Enter>");
23
24      // Crearea fluxurilor de caractere conectate la fluxurile de octeti
25      // obtinute de la socketul TCP
26      PrintStream outRetea = new
27                   PrintStream(conexiuneTCP.getOutputStream());
28      BufferedReader inRetea = new BufferedReader(
29                   new InputStreamReader(conexiuneTCP.getInputStream()));
30
31      while (true) {
32        // Citirea unei linii de la consola de intrare
33        System.out.print("Se trimite: ");
34        String mesajTrimis = inConsola.readLine();
35
36        // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
37        outRetea.println(mesajTrimis);
38        outRetea.flush();
39
40        // Citirea unei linii din fluxul de intrare TCP
41        String mesajPrimit = inRetea.readLine();
42
43        // Afisarea liniei citite la consola de iesire
44        System.out.println("S-a primit: " + mesajPrimit);
45
46        // Testarea conditiei de oprire a clientului
47        if (mesajPrimit.equals(".")) break;
48      }
49
50      // Inchiderea socketului (si implicit a fluxurilor)
51      conexiuneTCP.close();
52      System.out.println("Bye!");
53    }
54  }
```

5.3.5. Servere flux (TCP) non-concurente

In cazul serverelor care pot trata un singur client, codul necesar pentru stabilirea conexiunii si **pentru comunicatia cu clientul** este urmatorul:

```
// Initializare portServer
// Crearea socketului server (care accepta conexiunile)

ServerSocket serverTCP = new ServerSocket(portServer);

// Blocare in asteptarea cererii de conexiune

Socket conexiuneTCP = serverTCP.accept();

// Crearea fluxurilor conectate la fluxurile obtinute de la socketul TCP
PrintStream out = new PrintStream(conexiuneTCP.getOutputStream());
BufferedReader in = new BufferedReader(
               new InputStreamReader(conexiuneTCP.getInputStream()));

// Tratarea clientului
while (true) {
  // Citiri din fluxul de intrare TCP cu in.readLine();

  // Scrieri in fluxul de iesire TCP cu out.println(); out.flush();
} // Incheierea tratarii clientului

// Inchiderea socketului
conexiuneTCP.close();
```

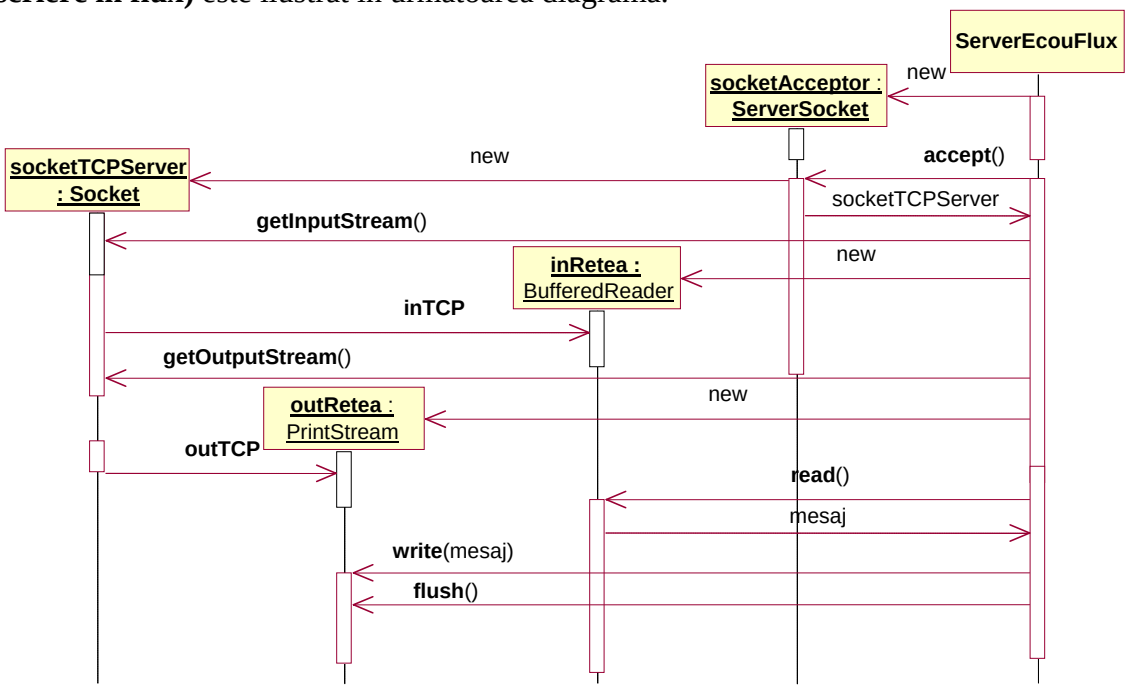
Secventa de mesaje schimbate la client, la server si intre client si server este, in acest caz, urmatoarea:



Urmatorul program este un **server ecou TCP care permite clientului trimiterea unui singur mesaj.**

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4    * Server ecou flux pentru servirea unui client o singura data
 5    */
 6  public class ServerEcouFlux {
 7
 8    public static void main (String args[]) throws IOException {
 9      BufferedReader inConsola  = new BufferedReader(new
10                                      InputStreamReader(System.in));
11
12      System.out.print("Introduceti numarul de port al serverului: ");
13      int portServer = Integer.parseInt(inConsola.readLine());
14
15      // Crearea socketului server (care accepta conexiunile)
16      ServerSocket serverTCP = new ServerSocket(portServer);
17      System.out.println("Server in asteptare pe portul "+portServer+"...");
18
19      // Blocare in asteptarea cererii de conexiune - in momentul acceptarii
20      // cererii se creaza socketul care serveste conexiunea
21      Socket conexiuneTCP = serverTCP.accept();
22      System.out.println("Conexiune TCP pe portul " + portServer + "...");
23
24      // Crearea fluxurilor de caractere conectate la fluxurile de octeti
25      // obtinute de la socketul TCP
26      PrintStream outRetea = new PrintStream(conexiuneTCP.getOutputStream());
27      BufferedReader inRetea = new BufferedReader(
28                      new InputStreamReader(conexiuneTCP.getInputStream()));
29
30      // Citirea unei linii din fluxul de intrare TCP
31      String mesajPrimit = inRetea.readLine();
32
33      // Afisarea liniei citite la consola de iesire
34      System.out.println("Mesaj primit: " + mesajPrimit);
35
36      // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
37      outRetea.println(mesajPrimit);
38      outRetea.flush();
39
40      // Inchiderea socketului (si implicit a fluxurilor)
41      conexiuneTCP.close();
42    }
43  }
```

Schimbul de mesaje la server (creare *socket* server, *socket* tratare client si fluxuri, citire din flux si scriere in flux) este ilustrat in urmatoarea diagrama.

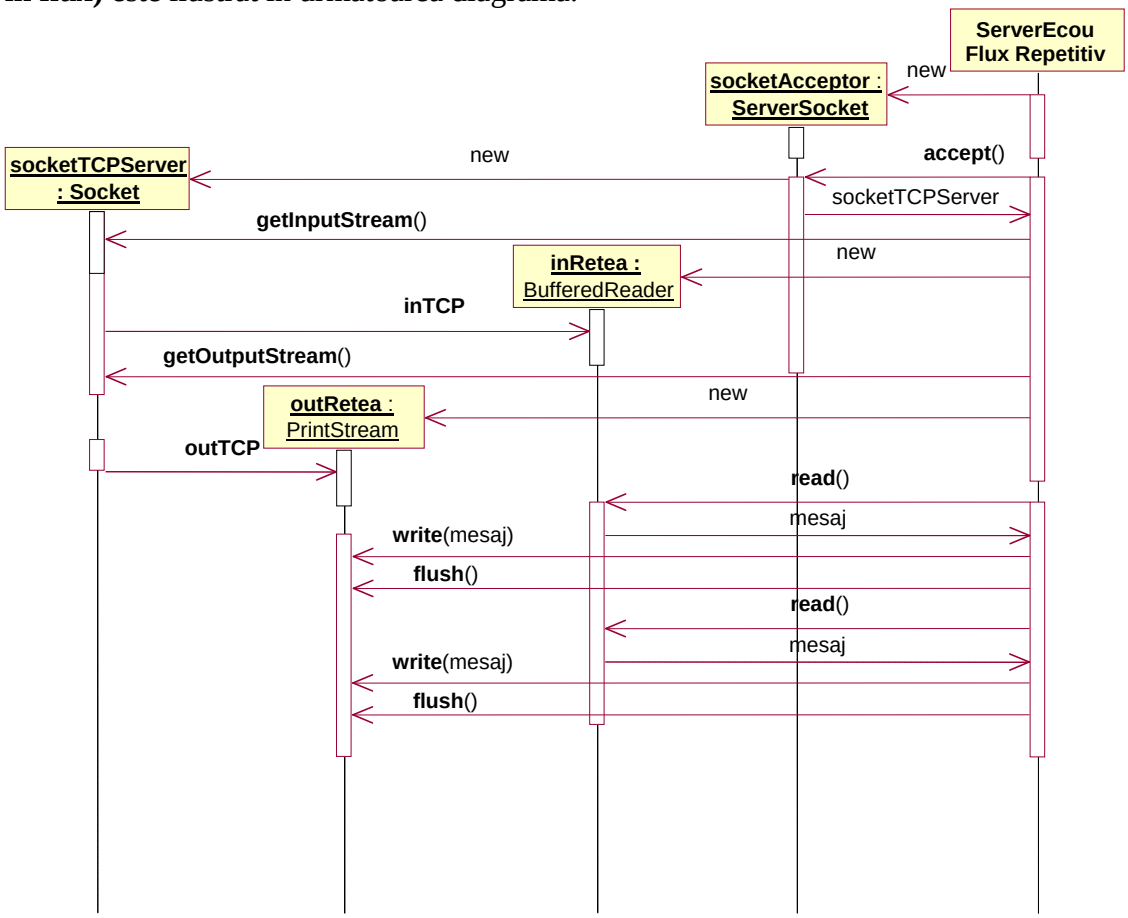


Urmatorul program este un **server ecou TCP care permite servirea unui singur client in mod repetat.**

Mesajul format dintr-un punct (".") semnaleaza serverului terminarea mesajelor de trimis, clientul urmand sa isi termine executia.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5    * Server ecou flux pentru servirea unui client in mod repetat
 6    */
 7  public class ServerEcouFluxRepetiv {
 8
 9    public static void main (String args[]) throws IOException {
10      BufferedReader inConsola  = new BufferedReader(new
11                                      InputStreamReader(System.in));
12
13      System.out.print("Introduceti numarul de port al serverului: ");
14      int portServer = Integer.parseInt(inConsola.readLine());
15
16      // Crearea socketului server (care accepta conexiunile)
17      ServerSocket serverTCP = new ServerSocket(portServer);
18      System.out.println("Server in asteptare pe portul "+portServer+"...");
19
20      // Blocare in asteptarea cererii de conexiune - in momentul acceptarii
21      // cererii se creaza socketul care serveste conexiunea
22      Socket conexiuneTCP = serverTCP.accept();
23      System.out.println("Conexiune TCP pe portul " + portServer + "...");
24
25      // Crearea fluxurilor de caractere conectate la fluxurile de octeti
26      // obtinute de la socketul TCP
27      PrintStream outRetea = new PrintStream(conexiuneTCP.getOutputStream());
28      BufferedReader inRetea = new BufferedReader(
29                      new InputStreamReader(conexiuneTCP.getInputStream()));
30
31      while (true) {
32        // Citirea unei linii din fluxul de intrare TCP
33        String mesajPrimit = inRetea.readLine();
34
35        // Afisarea liniei citite la consola de iesire
36        System.out.println("Mesaj primit: " + mesajPrimit);
37
38        // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
39        outRetea.println(mesajPrimit);
40        outRetea.flush();
41
42        // Testarea conditiei de oprire a servirii
43        if (mesajPrimit.equals(".")) break;
44      }
45
46      // Inchiderea socketului (si implicit a fluxurilor)
47      conexiuneTCP.close();
48      System.out.println("Bye!");
49    }
50  }
```

Schimbul de mesaje (creare *socket* server, *socket* tratare client si fluxuri, citire din flux si scriere in flux) este ilustrat in urmatoarea diagrama.



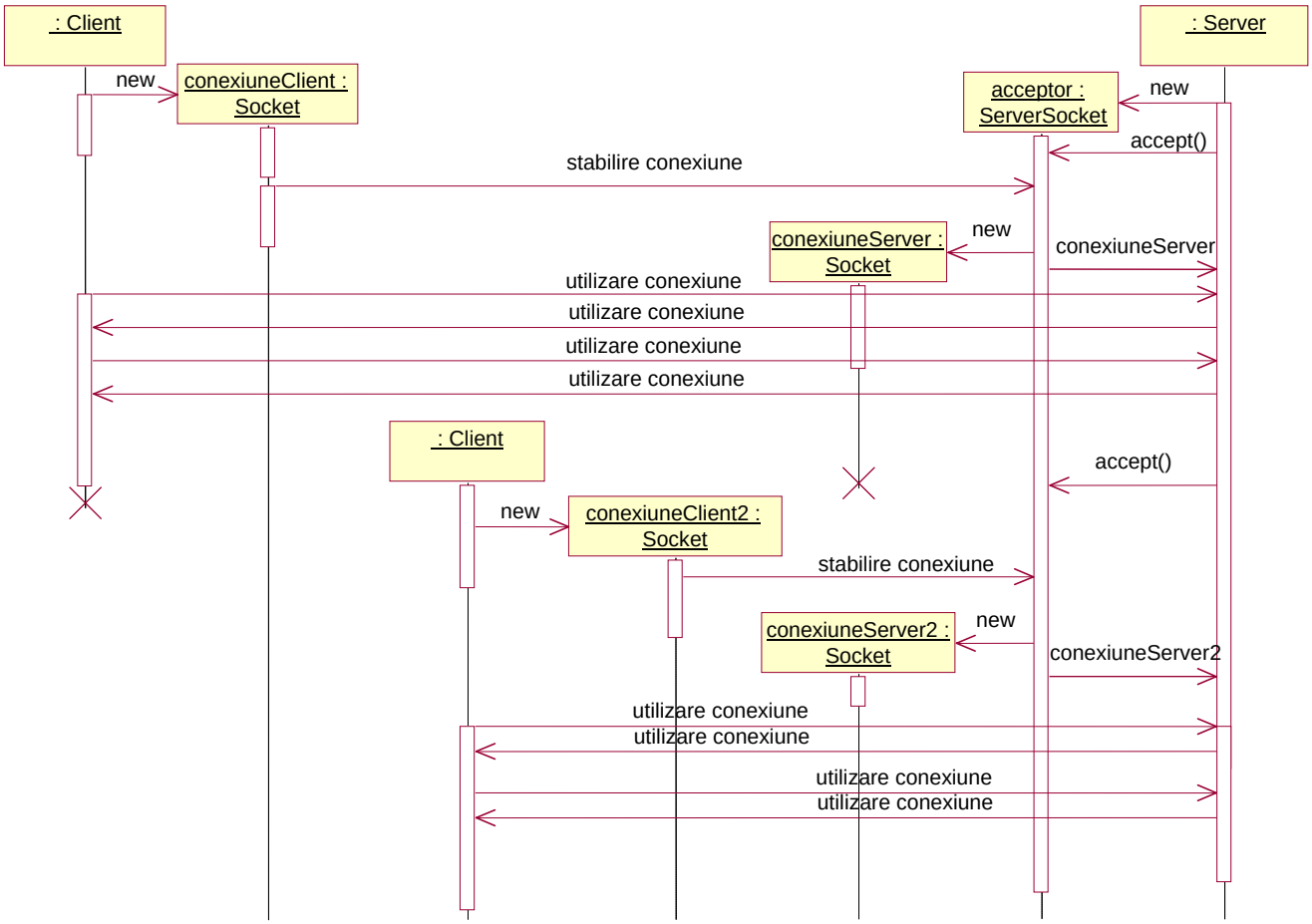
In cazul serverelor care pot trata mai multi clienti in mod secvential (succesiv), codul necesar pentru stabilirea conexiunilor si **pentru comunicatia cu clientii** este urmatorul:

```
// Initializare portServer
// Crearea socketului server (care accepta conexiunile)
ServerSocket serverTCP = new ServerSocket(portServer);

// Servirea mai multor clienti succesivi (in mod secvential)
while (true) {

  // Blocare in asteptarea cererii de conexiune

  Socket conexiuneTCP = serverTCP.accept();

  // Crearea fluxurilor conectate la fluxurile obtinute de la socketul TCP
  PrintStream out = new PrintStream(conexiuneTCP.getOutputStream());
  BufferedReader in = new BufferedReader(
              new InputStreamReader(conexiuneTCP.getInputStream()));

  // Tratarea clientului curent
  while (true) {
    // Citiri din fluxul de intrare TCP cu in.readLine();

    // Scrieri in fluxul de iesire TCP cu out.println(); out.flush();
  } // Incheierea tratarii clientului

  // Inchiderea socketului
  conexiuneTCP.close();

} // Gata pentru a servi urmatorul client
```

Server care permite receptia si trimiterea in ecou a mai multor mesaje sosite de la mai multi clienti, succesiv:

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4    * Server ecou flux pentru servirea mai multor clienti succesivi
 5    */
 6  public class ServerEcouFluxRepetivSecvential {
 7
 8    public static void main (String args[]) throws IOException {
 9      BufferedReader inConsola  = new BufferedReader(new
10                                      InputStreamReader(System.in));
11      System.out.print("Introduceti numarul de port al serverului: ");
12      int portServer = Integer.parseInt(inConsola.readLine());
13
14      // Crearea socketului server (care accepta conexiunile)
15      ServerSocket serverTCP = new ServerSocket(portServer);
16
17      // Servirea mai multor clienti succesivi (in mod secvential)
18      while (true) {
19        System.out.println("Server in asteptare pe port "+portServer+"...");
20         // Blocare in asteptarea cererii de conexiune - in momentul
21        // acceptarii cererii se creaza socketul care serveste conexiunea
22        Socket conexiuneTCP = serverTCP.accept();
23        System.out.println("Conexiune TCP pe portul " + portServer + "...");
24
25        // Crearea fluxurilor de caractere conectate la fluxurile de octeti
26        // obtinute de la socketul TCP
27        PrintStream outRetea = new
28               PrintStream(conexiuneTCP.getOutputStream());
29        BufferedReader inRetea = new BufferedReader(
30               new InputStreamReader(conexiuneTCP.getInputStream()));
31
32        // Servirea clientului curent
33        while (true) {
34          // Citirea unei linii din fluxul de intrare TCP
35          String mesajPrimit = inRetea.readLine();
36
37          // Afisarea liniei citite la consola de iesire
38          System.out.println("Mesaj primit: " + mesajPrimit);
39
40          // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
41          outRetea.println(mesajPrimit);
42          outRetea.flush();
43
44          // Testarea conditiei de oprire a servirii
45          if (mesajPrimit.equals(".")) break;
46        }
47        // Inchiderea socketului (si implicit a fluxurilor)
48        conexiuneTCP.close();
49        System.out.println("Bye!");
50      }
51    }
52  }
```

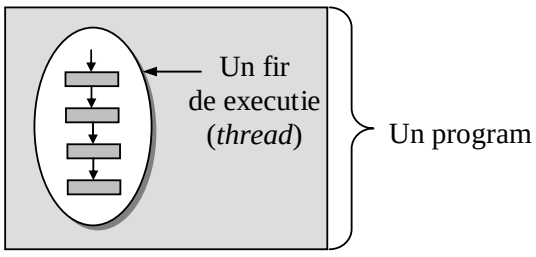
Secventa de mesaje schimbate la client, la server si intre client si server este urmatoarea:



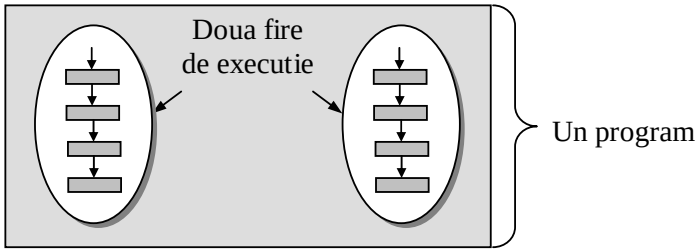
5.3.6. Fire de executie (threads)

Programele de calcul simple sunt **secventiale**, fiecare avand **un inceput, o secventa de executii si un sfarsit**. In orice moment pe durata executiei unui astfel de program **exista un singur punct de executie**.

Un fir de executie (*thread*), sau mai simplu, **un fir**, este similar acestor programe secventiale, in sensul ca are **un inceput, o secventa de executii si un sfarsit**, si in orice moment pe durata executiei firului **exista un singur punct de executie**. Totusi, **un fir nu este el insusi un program**, deoarece **nu poate fi executat de sine statator**. In schimb, **firul este executat** (ruleaza) **intr-un program**. Relatia dintre un program si un fir de executie este ilustrata mai jos.



Un fir de executie **(*thread*)** poate fi definit ca **un flux de control secvential in cadrul unui program**. **Posibilitatea utilizarii mai multor fire de executie intr-un singur program**, ruland (fiind executate) in acelasi timp si realizand diferite sarcini (nu in mod necesar diferite), este numita ***multithreading***. Un navigator (*browser*) Web este un exemplu de **aplicatie multi-filara (*multithreaded*)**. In *browser* **se poate parcurge pagina in timpul descarcarii unei miniaplicatii Java (*applet*)** a unei imagini, etc.



O denumire alternativa a firelor de executie este de ***lightweight process***, deoarece un fir este similar unui proces real in sensul ca ambele sunt fluxuri de control secventiale. Totusi, **un fir** este considerat *lightweight* deoarece el **ruleaza in contextul unui program si are la dispozitie reursele alocate pentru acel program si mediul programului**.

Ca flux de control secvential, **un fir de executie utilizeaza o parte din resursele programului in care ruleaza**. De exemplu, el trebuie sa aiba propria stiva de executie si propriul contor de program. Codul firului de executie lucreaza doar in acel context. De aceea, **o denumire alternativa** este aceea de **context de executie**.

Masina virtuala Java, JVM (*Java Virtual Machine*), **permite aplicatiilor sa aiba mai multe fire de executie care ruleaza in mod concurent** (in paralel).

Fiecare fir de executie are un anumit **nivel de prioritate**. Firele cu nivel de prioritate mai mare sunt executate inaintea celor cu nivel de prioritate mai mic. Atunci cand codul care ruleaza intr-un fir de executie creaza un nou obiect de tip `Thread`, noul fir de executie are initial acelasi nivel de prioritate cu firul de executie care l-a creat.

Fiecare fir de executie poate fi marcat ca *daemon*. Un fir de executie creat de un fir de executie *daemon* este la randul lui un *daemon*.

La lansarea masinii virtuale Java, exista in mod normal un singur fir de executie *non-daemon* (cel ce apeleaza metoda `main()` a clasei cu care incepe executia). **Masina virtuala Java continua sa execute firele pana cand**:
- **este apelata metoda `exit()`** a clasei `Runtime` si managerul de securitate permite executia operatiei exit, sau
- **toate firele care nu sunt *daemoni* au murit** (si-au incheiat executia), fie prin returnarea din apelul metodei `run()`, fie prin aruncarea unei exceptii care se propaga dincolo de metoda `run()`.

Pentru a crea un nou fir de executie exista doua modalitati.

1. Se poate declara o clasa ca subclasa a clasei **`Thread`**, subclasa care **trebuie sa rescrie codul (*override*) metodei `run()` a clasei `Thread`** (care nu contine nici un cod), noul fir de executie fiind creat prin alocarea si lansarea unei instante a subclasei.

Formatul pentru crearea unei subclase care extinde clasa `Thread` si ii reimplementeaza metoda `run()` este urmatorul:

```
class FirT extends Thread {

  public void run() {
    // codul firului de executie
  }
}
```

Formatul pentru crearea unei instante a subclasei este urmatorul:

```
FirT fir = new FirT();
```

2. Ca alternativa, se poate declara o clasa care implementeaza interfata **`Runnable`**, interfata care contine doar declaratia unei metode **`run()`** (declaratie identical celei din clasa `Thread`, deoarece clasa `Thread` implementeaza interfata `Runnable`), noul fir de executie fiind creat prin alocarea unei instante a noii clase, pasarea acestei instante ca parametru al constructorului, la crearea unei instante a clasei `Thread`, si lansarea acelei instante a clasei `Thread`.

Formatul pentru crearea unei clase care implementeaza interfata `Runnable` si ii implementeaza metoda `run()` este urmatorul:

```
class FirR implements Runnable {

  public void run() {
    // codul firului de executie
  }
}
```

Formatul pentru crearea unei instante a noii clase si apoi a unei instante a clasei `Thread` este urmatorul:

```
FirR r = new FirR();
Thread fir = new Thread(r);
```

In ambele cazuri **formatul pentru lansarea noului fir de executie,** este urmatorul:

```
fir.start();
```

Desigur, exista si **variante compacte pentru crearea si lansarea noilor fire de executie**, cum ar fi:

```
new FirT().start();                 // nu exista variabila de tip FirT
                                    // care sa refere explicit firul
```

sau

```
FirR r = new FirR();
new Thread(r).start();              // nu exista variabila de tip Thread
                                    // care sa refere explicit firul
```

sau

```
Thread fir = new Thread(new FirR());
fir.start();                        // nu exista variabila de tip FirR
```

sau

```
new Thread(new FirR()).start();     // nu exista variabila de tip Thread
                                    // care sa refere explicit firul
                                    // si nici variabila de tip FirR
```

Fiecare fir de executie are un nume, cu scopul identificarii lui. Mai multe fire de executie pot avea acelasi nume. **Daca nu este specificat** un nume in momentul constructiei (initializarii) firului, **un nou nume (aleator stabilit) este generat** pentru acesta.

5.3.7. Clasa Thread – interfata publica

1. Principalii constructori ai clasei **`Thread`:**

`Thread``()` Initializeaza un nou obiect de tip `Thread` (care trebuie sa implementeze metoda `run()`)
`Thread``(Runnable target)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` (care trebuie sa implementeze metoda `run()`) dintr-o clasa care implementeaza interfata `Runnable`.
`Thread``(Runnable target, String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` dintr-o clasa care implementeaza interfata `Runnable`, si un nume `name`.

`Thread``(String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un nume `name`.
`Thread``(ThreadGroup group, Runnable target)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` dintr-o clasa care implementeaza interfata `Runnable`, grupul de fire de care apartine `group`.
`Thread``(ThreadGroup group, Runnable target, String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` dintr-o clasa care implementeaza interfata `Runnable`, grupul de fire de care apartine `group`, si un nume.
`Thread``(ThreadGroup group, Runnable target, String name, long stackSize)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta dintr-o clasa care implementeaza interfata `Runnable`, grupul de fire de care apartine, un nume, si dimensiunea stivei care ii va fi alocata pentru executie (*stack size*).
`Thread``(ThreadGroup group, String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica grupul de fire de care apartine `group`, si un nume `name`.

2. Declaratiile si descrierea catorva metode ale clasei **`Thread`**:

`int`	**`activeCount`**`()` Returneaza numarul firelor de executie active din grupul curent de fire.
`void`	**`checkAccess`**`()` Determina daca firul curent executat are permisiunea de a modifica firul de executie curent (`this`).
`static Thread`	**`currentThread`**`()` Returneaza o referinta catre obiectul `Thread` curent executat.
`void`	**`destroy`**`()` Distruge firul de executie curent (`this`), fara eliberarea resurselor.
`static void`	**`dumpStack`**`()` Afiseaza informatiile din stiva ale firului de executie curent.
`static int`	**`enumerate`**`(Thread[] tarray)` Copiaza in tabloul specificat ca parametru referinte catre toate firele de executie active din grupul de fire curent si din subgrupurile sale.
`ClassLoader`	**`getContextClassLoader`**`()` Returneaza obiectul context al firului de executie curent.
`String`	**`getName`**`()` Returneaza numele firului de executie curent.
`int`	**`getPriority`**`()` Returneaza nivelul de prioritate al firului de executie curent.
`ThreadGroup`	**`getThreadGroup`**`()` Returneaza grupul de fire caruia ii apartine firul de executie curent.
`static boolean`	**`holdsLock`**`(Object obj)` Returneaza `true` daca firul de executie curent detine lock-ul monitorului creat pe obiectul specificat ca parametru.
`void`	**`interrupt`**`()` Intrerupe firul de executie curent.
`static boolean`	**`interrupted`**`()` Verifica daca firul de executie curent executat a fost intrerupt.
`boolean`	**`isAlive`**`()` Verifica daca firul de executie curent este in viata.

boolean	**isDaemon**() Verifica daca firul de executie curent este fir *daemon*.
boolean	**isInterrupted**() Verifica daca firul de executie curent a fost intrerupt.
void	**join**() Asteapta pana cand firul de executie curent moare.
void	**join**(long millis) Asteapta cel putin valoarea specificata ca parametru, in milisecunde, pana cand firul de executie curent moare.
void	**join**(long millis, int nanos) Asteapta cel putin `millis` milisecunde plus `nanos` nanosecunde pana cand firul de executie curent moare.
void	**run**() Implicit metoda nu executa nimic si returneaza.
void	**setContextClassLoader**(ClassLoader cl) Stabileste obiectul context al firului de executie curent.
void	**setDaemon**(boolean on) Marcheaza firul de executie curent ca fir *daemon*.
void	**setName**(String name) Schimba numele firului de executie curent cu cel specificat ca parametru.
void	**setPriority**(int newPriority) Stabileste nivelul de prioritate al firului de executie curent.
static void	**sleep**(long millis) Forteaza firul de executie curent executat sa cedeze temporar executia, pentru numarul de milisecunde specificat.
static void	**sleep**(long millis, int nanos) Forteaza firul de executie curent executat sa cedeze temporar executia, pentru numarul de milisecunde plus numarul de nanosecunde specificat.
void	**start**() Forteaza firul de executie curent sa isi inceapa executia. Masina virtuala Java apeleaza metoda `run()` a firului de executie curent.
void	**stop**() **Deprecated.** *Metoda este in mod inerent nesigura. In majoritatea cazurilor utilizarea `stop()` poate fi inlocuita cu un cod care modifica o anumita variabila care indica faptul ca firul de executie tinta trebuie sa isi incheie executia, firul de executie tinta urmand sa verifice variabila in mod regulat si sa returneze din metoda sa run() atunci cand variabila indica necesitatea de a se incheia executia. Daca firul de executie tinta asteapta pentru perioade lungi de timp, atunci trebuie utilizata metoda `interrupt()` pentru a intrerupe asteptarea.*
void	**stop**(Throwable obj) **Deprecated.** *Vezi explicatia de mai sus*
String	**toString**() Returneaza o reprezentare ca sir de caractere a firului de executie curent, incluzand numele, prioritatea, si grupul de fire ale firului de executie curent.
static void	**yield**() Forteaza firul de executie curent executat sa se opreasca temporar si sa permita altui fir de executie sa fie executat.

Urmatoarea clasa Java, **FirSimplu**, extinde clasa **Thread**, iar metoda ei principala lanseaza metoda **run**() ca nou fir de executie.

```
 1    public class FirSimplu extends Thread {
 2      public FirSimplu(String str) {
 3          super(str);
 4      }
 5      public void run() {
 6          for (int i = 0; i < 10; i++) {
 7              System.out.println(i + " " + getName());
 8              try {
 9                  sleep((long)(Math.random() * 1000));
10              } catch (InterruptedException e) {}
11          }
12          System.out.println("Gata! " + getName());
13      }
14
15      public static void main (String[] args) {
16          new FirSimplu("Unu").start();
17      }
18    }
```

Rezultatul pe ecran al executiei programului:

```
>java FirSimplu
0 Unu
1 Unu
2 Unu
3 Unu
4 Unu
Gata! Unu
```

Clasa **DemoDouaFire** lanseaza doua fire de executie **FirSimplu** care sunt executate concurent.

```
1  public class DemoDouaFire {
2
3      public static void main (String[] args) {
4          new FirSimplu("Unu").start();
5          new FirSimplu("Doi").start();
6      }
7  }
```

Rezultatele pe ecran a doua executii succesive:

```
>java DemoDouaFire
0 Unu
0 Doi
1 Unu
1 Doi
2 Unu
2 Doi
3 Doi
3 Unu
4 Doi
Gata! Doi
4 Unu
Gata! Unu
```

```
>java DemoDouaFire
0 Unu
0 Doi
1 Unu
1 Doi
2 Unu
2 Doi
3 Unu
4 Unu
Gata! Unu
3 Doi
4 Doi
Gata! Doi
```

Se observa ca firele pot avea evolutii diferite, in functie de durata intarzierii introdusa de linia de cod:

```
sleep((long)(Math.random() * 1000));
```

a programului **FirSimplu**.

5.3.8. Servere flux concurente

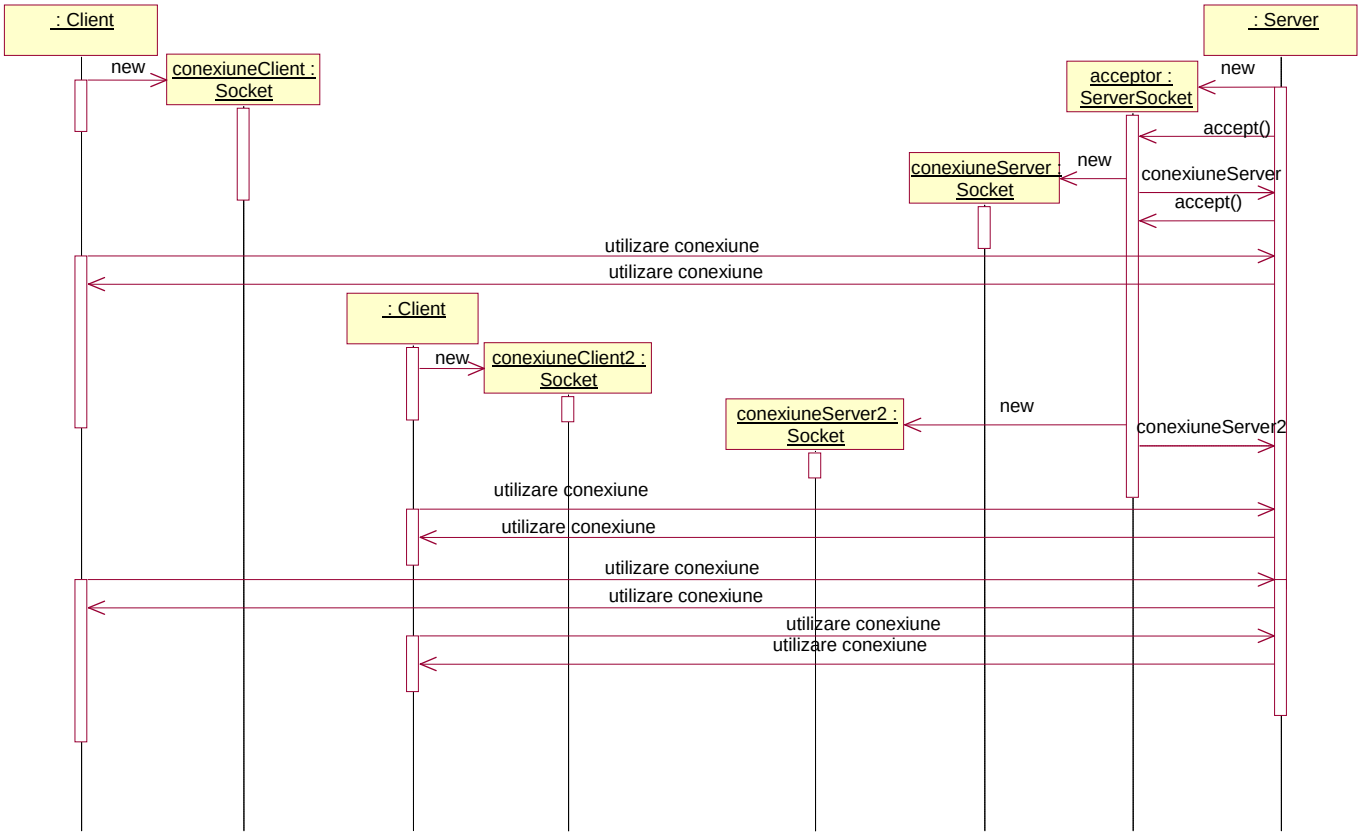
In cazul serverelor care pot trata mai multi clienti in mod concurent (in paralel), codul necesar pentru stabilirea conexiunilor cu clientii, ce urmeaza a fi tratati in clasa fir de executie pe care o vom denumi in mod generic **`ClasaFiruluiDeTratare`**, este urmatorul:

```
// initializare portServer

// Crearea socketului server (care accepta conexiunile)

ServerSocket serverTCP = new ServerSocket(portServer);

// Servirea mai multor clienti in paralel (in mod concurent)
while (true) {

  // Blocare in asteptarea cererii de conexiune

  Socket conexiuneTCP = serverTCP.accept();

  // Crearea si lansarea firului de executie pentru tratarea noului client

  ClasaFiruluiDeTratare firExecutie = new ClasaFiruluiDeTratare(conexiuneTCP);
  firExecutie.start();

} // Gata pentru a accepta urmatorul client
```

Codul necesar pentru comunicatia cu un client, din clasa fir de executie pentru tratarea clientului (denumita in mod generic **`ClasaFiruluiDeTratare`**), este urmatorul:

```
Socket socketTCP;                          // Atribut

public ClasaFiruluiDeTratare(Socket s) {   // Constructor
  socketTCP = s;                           // Initializarea atributului
}

public void run() {                       // Implementarea firului de executie

  // Crearea fluxurilor conectate la fluxurile obtinute de la socketul TCP

  PrintStream out = new PrintStream(conexiuneTCP.getOutputStream());
  BufferedReader in = new BufferedReader(
              new InputStreamReader(conexiuneTCP.getInputStream()));

  // Tratarea clientului curent
  while (true) {
    // Citiri din fluxul de intrare TCP cu in.readLine();

    // Scrieri in fluxul de iesire TCP cu out.println(); out.flush();
  } // Incheierea tratarii clientului

  // Inchiderea socketului
  conexiuneTCP.close();

}
```

Secventa de mesaje schimbate la client, la server si intre client si server este, in acest caz, urmatoarea:



Exemplu complet de server ecou care poate trata mesaje sosite de la mai multi clienti in paralel:

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5    * Server ecou flux pentru servirea mai multor clienti in acelasi timp
 6    */
 7  public class ServerEcouFluxRepetivConcurent extends Thread {
 8
 9    /**
10      * Socket de servire client
11      */
12    private Socket conexiuneTCP;
13
14    /**
15      * Initializare socket servire client
16      */
17    public ServerEcouFluxRepetivConcurent(Socket socketTCP) {
18
19      conexiuneTCP = socketTCP;
20    }
21
22    /**
23      * Punct de intrare program
24      */
25    public static void main (String args[]) throws IOException {
26
27      BufferedReader inConsola  = new BufferedReader(new
28                                      InputStreamReader(System.in));
29      System.out.print("Introduceti numarul de port al serverului: ");
```

```
30
31      int portServer = Integer.parseInt(inConsola.readLine());
32
33      // Crearea socketului server (care accepta conexiunile)
34      ServerSocket serverTCP = new ServerSocket(portServer);
35
36      System.out.println("Server in asteptare pe portul "+portServer+"...");
37
38      // Servirea mai multor clienti in acelasi timp (in mod concurent)
39      while (true) {
40
41        // Blocare in asteptarea cererii de conexiune - in momentul
42        // acceptarii cererii se creaza socketul care serveste conexiunea
43        Socket socketTCP = serverTCP.accept();
44        System.out.println("Conexiune TCP pe portul " + portServer + "...");
45
46        new ServerEcouFluxRepetivConcurent(socketTCP).start(); // run()
47
48      }
49    }
50
51    /**
52      * Fir de servire client
53      */
54    public void run() {
55
56      try {
57
58        // Crearea fluxurilor de caractere conectate la fluxurile de octeti
59        // obtinute de la socketul TCP
60        PrintStream outRetea = new
61              PrintStream(conexiuneTCP.getOutputStream());
62        BufferedReader inRetea = new BufferedReader(
63              new InputStreamReader(conexiuneTCP.getInputStream()));
64
65        BufferedReader inConsola  = new BufferedReader(new
66                                      InputStreamReader(System.in));
67
68        // Servirea clientului curent
69        while (true) {
70          // Citirea unei linii din fluxul de intrare TCP
71          String mesajPrimit = inRetea.readLine();
72
73          // Afisarea liniei citite la consola de iesire
74          System.out.println("Mesaj primit: " + mesajPrimit);
75
76          // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
77          outRetea.println(mesajPrimit);
78          outRetea.flush();
79
80          // Testarea conditiei de oprire a servirii
81          if (mesajPrimit.equals(".")) break;
82        }
83
84        // Inchiderea socketului (si implicit a fluxurilor)
85        conexiuneTCP.close();
86        System.out.println("Bye!");
87      }
88      catch (IOException ex) {
89        System.err.println(ex);
90      }
91    }
92  }
```

5.4. Socketuri datagrama (UDP)

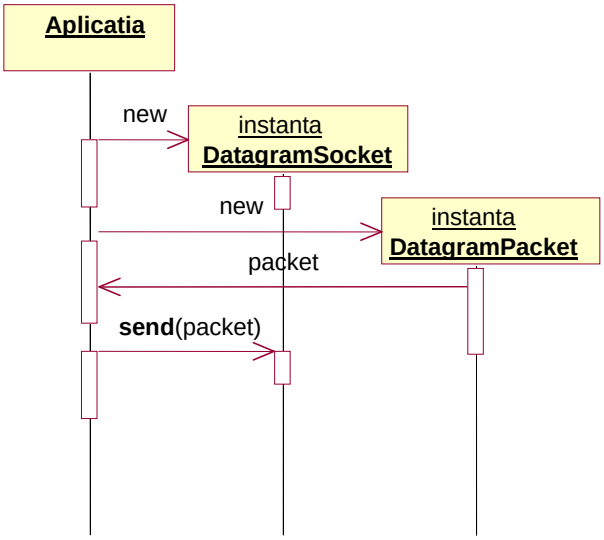
5.4.1. Lucrul cu socketuri datagrama (UDP)

Java ofera, in pachetul **`java.net`**, mai multe clase pentru lucrul cu *socket-uri* datagrame (UDP). Urmatoarele **clase Java** sunt **implicate in realizarea comunicatiilor UDP** obisnuite: **`DatagramSocket`** si **`DatagramPacket`**.

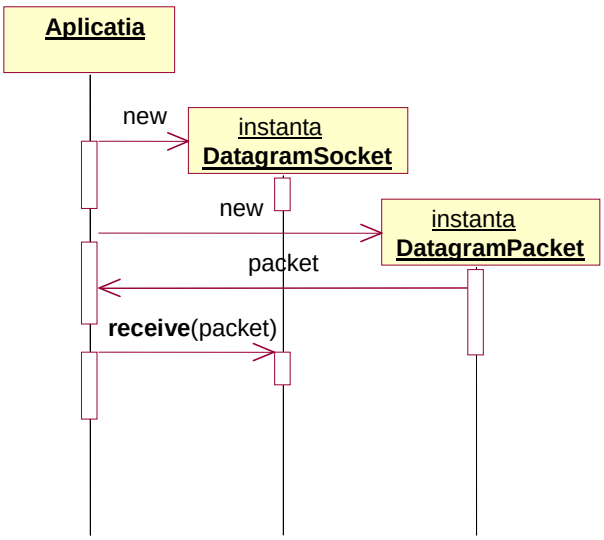
Clasa **`DatagramPacket`** reprezinta un **pachet UDP** (o **datagrama**). Pachetele datagrama sunt utilizate pentru livrare fara conexiune si includ in mod normal informatii privind adresele IP si porturile sursa si destinatie.

Clasa **`DatagramSocket`** reprezinta ***socket-ul* UDP**, **prin care se trimit sau se primesc pachete datagrama** peste retele IP **prin intermediul protocolului UDP**.

Un **`DatagramPacket` este trimis printr-un `DatagramSocket` apeland** la metoda **`send()`** a clasei **`DatagramSocket`**, **pasand** ca parametru respectivul **`DatagramPacket`.**



Un **`DatagramPacket` este primit printr-un `DatagramSocket` apeland** la metoda **`receive()`** a clasei **`DatagramSocket`**, **pasand** ca parametru un **`DatagramPacket`** pregatit pentru receptie.



Clasa **`MulticastSocket`** poate fi utilizata pentru trimiterea si receptia unui `DatagramPacket` catre, respectiv dinspre, un **grup multicast**. Clasa `MulticastSocket` este o subclasa a `DatagramSocket` care adauga functionalitati legate de multicast.

5.4.2. Clasa pachet (datagrama) UDP (DatagramPacket) – interfata publica

1. Principalii constructori ai clasei `DatagramPacket`:

`DatagramPacket``(byte[] buf, int length)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **receptia** unui pachet de lungime `length`, furnizandu-i tabloul de octeti `buf` in care sa fie plasate datele pachetului (`length` trebuie sa fie mai mic sau egal cu `buf.length`).
`DatagramPacket``(byte[] buf, int offset, int length)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **receptia** unui pachet de lungime `length`, furnizandu-i tabloul de octeti `buf` in care sa fie plasate datele pachetului si specificandu-i indexul `offset` de la care sa inceapa plasarea datelor pachetului in tablou (`length+offset` trebuie sa fie mai mic sau egal cu `buf.length`).
`DatagramPacket``(byte[] buf, int length, InetAddress address, int port)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **trimiterea** unui pachet de lungime `length` catre numarul de port UDP specificat (`port`) al gazdei cu adresa specificata (`address`), furnizandu-i tabloul de octeti `buf` din care sa fie preluate datele pachetului (`length` trebuie sa fie mai mic sau egal cu `buf.length`).
`DatagramPacket``(byte[] buf, int offset, int length, InetAddress address, int port)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **trimiterea** unui pachet de lungime `length` catre numarul de port UDP specificat (`port`) al gazdei cu adresa specificata (`address`), furnizandu-i tabloul de octeti `buf` din care sa fie preluate datele pachetului si specificandu-i indexul `offset` de la care sa inceapa preluarea datelor pachetului din tablou (`length+offset` trebuie sa fie mai mic sau egal cu `buf.length`).

2. Declaratiile si descrierea catorva metode ale clasei **`DatagramPacket`**:

`InetAddress`	**`getAddress`**`()` Returneaza **adresa IP** sub forma de obiect `InetAddress` a masinii catre care pachetul curent va fi trimis sau de la care pachetul curent va fi receptionat.
`byte[]`	**`getData`**`()` Returneaza tabloul de octeti (*bufferul*) care contine **datele** pachetului curent.
`int`	**`getLength`**`()` Returneaza numarul de octeti (**lungimea** *bufferului*) de date de trimis sau de receptionat.
`int`	**`getOffset`**`()` Returneaza indexul (***offsetul***) la care sunt plasate datele de trimis sau de receptionat in tabloul de octeti.
`int`	**`getPort`**`()` Returneaza **numarul de port UDP** al masinii catre care pachetul curent va fi trimis sau de la care pachetul curent va fi receptionat.
`void`	**`setAddress`**`(InetAddress iaddr)` Stabileste **adresa IP** sub forma de obiect `InetAddress` a masinii catre care pachetul curent va fi trimis.

void	**setData**(byte[] buf) Stabileste tabloul de octeti (*bufferul*) care contine **datele** pachetului curent (implicit indexul `offset=0`).
void	**setData**(byte[] buf, int offset, int length) Stabileste tabloul de octeti (*bufferul*) care contine **datele** pachetului curent specificandu-i indexul `offset` de la care sa inceapa plasarea datelor pachetului in tablou.
void	**setLength**(int length) Stabileste numarul de octeti (**lungimea** *bufferului*) de date de trimis sau de receptionat.
void	**setPort**(int iport) Stabileste **numarul de port UDP** al masinii catre care pachetul curent va fi trimis.

In continuare este prezentata **secventa tipica pentru crearea unui pachet datagrama (UDP) pentru timitere**:

```
// Stabilirea adresei serverului
String adresaIP = "localhost";

// Stabilirea portului serverului
int portUDP = 2000;

// Crearea tabloului de octeti (bufferului) de date
String mesaj = "mesaj de trimis";
byte[] bufferDate = mesaj.getBytes();

// Crearea pachetului de trimis
try {
  DatagramPacket pachetUDP = new DatagramPacket(bufferDate,
        bufferDate.length, InetAddress.getByName(adresaIP), portUDP);
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
```

In continuare este prezentata **secventa tipica pentru crearea unui pachet datagrama (UDP) pentru receptie**:

```
// Crearea tabloului de octeti (bufferului) de date
byte[] bufferDate = new byte[4096];

// Crearea pachetului de primit
try {
  DatagramPacket pachetUDP = new DatagramPacket(bufferDate,
                                                bufferDate.length);
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
```

ca si **secventa tipica pentru citirea din pachetul UDP a datelor primite**:

```
// Obtinerea datelor pachetului ca tablou de octeti
bufferDate = pachetUDP.getData();

// Reconstructia mesajului text
String mesajPrimit =
             new String(bufferDate, 0, pachetUDP.getLength());
```

In continuare este prezentata o **aplicatie** care permite **obtinerea si afisarea informatiilor privind un pachet creat pentru trimitere**.

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Afisare informatii privind pachet UDP (datagrama)
 5   */
 6  public class InfoPachetUDP {
 7
 8    /**
 9     *  Afiseaza informatii privind pachetul UDP specificat ca parametru
10     */
11    public static void afisareInfoPachetUDP(DatagramPacket datagrama,
12                                            boolean deTrimis) {
13      System.out.println("\n -------------------------------------");
14      System.out.print("\n Pachetul UDP ");
15
16      if (deTrimis) System.out.print("adresat " + datagrama.getAddress() +
17                       " pe portul " + datagrama.getPort());
18
19      else  System.out.print("de la adresa " + datagrama.getAddress() +
20                " si portul " + datagrama.getPort());
21
22      System.out.println(" contine " + datagrama.getLength() +
23          " octeti de date:\n" + new String(datagrama.getData()));
24    }
25
26    /**
27     *  Testarea si exemplificarea modului de utilizare
28     */
29    public static void main(String args[]) throws IOException {
30      BufferedReader inConsola  = new BufferedReader(new
31                                      InputStreamReader(System.in));
32      System.out.print("Introduceti adresa IP dorita: ");
33      String adresaIP = inConsola.readLine();
34
35      System.out.print("Introduceti numarul de port dorit: ");
36      int portUDP = Integer.parseInt(inConsola.readLine());
37
38      System.out.print("Introduceti continutul pachetului: ");
39      byte[] bufferDate = inConsola.readLine().getBytes();
40      try {
41        DatagramPacket pachetUDP = new DatagramPacket(bufferDate,
42              bufferDate.length, InetAddress.getByName(adresaIP), portUDP);
43
44        afisareInfoPachetUDP(pachetUDP, true);
45      }
46      catch (UnknownHostException ex) {
47        System.err.println(ex);
48      }
49    }
50  }
```

5.4.3. Clasa socket UDP (DatagramSocket) – interfata publica

1. Principalii constructori ai clasei `DatagramSocket`:

`DatagramSocket``()` Construieste un *socket* datagrame si il leaga la un port UDP disponibil pe masina locala.
`DatagramSocket``(int port)` Construieste un *socket* datagrame si il leaga la portul UDP specificat pe masina locala.
`DatagramSocket``(int port, InetAddress laddr)` Construieste un *socket* datagrame, legat pe masina locala la portul UDP si adresa IP specificate.

2. Declaratiile si descrierea catorva metode ale clasei **`DatagramPacket`**:

`InetAddress`	**`getLocalAddress`**`()` Obtine adresa IP locala la care este legat *socketul* curent.
`int`	**`getLocalPort`**`()` Returneaza numarul de port local la care este legat *socketul* curent.
`boolean`	**`isBound`**`()` Returneaza o valoare logica indicand daca *socketul* curent este legat cu succes la o adresa locala.
`void`	**`receive`**`(DatagramPacket p)` Receptioneaza un pachet datagrama prin *socketul* curent.
`void`	**`send`**`(DatagramPacket p)` Trimite un pachet datagrama prin *socketul* curent.
`void`	**`close`**`()` Inchide *socketul* datagrama curent.
`boolean`	**`isClosed`**`()` Returneaza o valoare logica indicand daca *socketul* curent este inchis.
`void`	**`connect`**`(InetAddress address, int port)` Conecteaza *socketul* curent la adresa IP si numarul de port specificate (acestea actioneaza ca un filtru, prin *socketul* conectat putand fi trimise sau primite pachete doar catre respectiv de la adresa IP si numarul de port specificate). Implicit *socketul* este neconectat.
`InetAddress`	**`getInetAddress`**`()` Returneaza adresa IP la care *socketul* curent este conectat (`null` daca nu este conectat).
`int`	**`getPort`**`()` Returneaza numarul de port UDP la care *socketul* curent este conectat (`-1` daca nu este conectat).
`void`	**`disconnect`**`()` Deconecteaza *socketul* curent.
`boolean`	**`isConnected`**`()` Returneaza o valoare logica indicand daca *socketul* curent este conectat.
`boolean`	**`getBroadcast`**`()` Returneaza o valoare logica indicand daca SO_BROADCAST este validat.
`void`	**`setBroadcast`**`(boolean on)` Valideaza/invalideaza SO_BROADCAST.

`int`	**`getSoTimeout`**`()` Obtine valoarea optiunii SO_TIMEOUT.
`void`	**`setSoTimeout`**`(int timeout)` Stabileste valoarea optiunii SO_TIMEOUT la un *timeout* specificat, in millisecunde.
`int`	**`getReceiveBufferSize`**`()` Returneaza valoarea optiunii SO_RCVBUF pentru *socketul* curent, adica dimensiunea *bufferului* utilizat de platforma pentru pachetele primite de *socketul* curent.
`void`	**`setReceiveBufferSize`**`(int size)` Stabileste optiunea SO_RCVBUF pentru *socketul* curent la valoarea specificata.
`int`	**`getSendBufferSize`**`()` Returneaza valoarea optiunii SO_ SNDBUF pentru *socketul* curent, adica dimensiunea *bufferului* utilizat de platforma pentru pachetele trimise de *socketul* curent.
`void`	**`setSendBufferSize`**`(int size)` Stabileste optiunea SO_SNDBUF pentru *socketul* curent la valoarea specificata.
`int`	**`getTrafficClass`**`()` Obtine valoarea octetului clasa de trafic (QoS) sau tip de serviciu (ToS) in antetul datagramei IP care incapsuleaza datagramele UDP trimise prin *socketul* curent.
`void`	**`setTrafficClass`**`(int tc)` Stabileste valoarea octetului clasa de trafic (QoS) sau tip de serviciu (ToS) in antetul datagramei IP care incapsuleaza datagramele UDP trimise prin *socketul* curent.

In continuare este prezentata **secventa tipica pentru crearea *socket-ului* unei aplicatii client**:

```
// Crearea socketului
DatagramSocket socketUDP = new DatagramSocket();
```

ca si **pentru crearea *socket-ului* unei aplicatii server**:

```
// Stabilirea portului serverului
int portServer = 2000;

// Crearea socketului
DatagramSocket socketUDP = new DatagramSocket(portServer);
```

Socket-ul UDP poate fi utilizat pentru **trimiterea de date**:

```
// Trimiterea pachetului UDP
socketUDP.send(pachetDeTrimis);
```

sau pentru **primirea de date**:

```
// Receptia unui pachet UDP - blocare in asteptare
socketUDP.receive(pachetPrimit);
```

Dupa utilizare, *socket-ul* poate fi **inchis**:

```
// Inchiderea socketului
socketUDP.close();
```

Ca si in cazul *socket-urilor* flux, metoda **`setTrafficClass`**`(int tc)` **stabileste clasa de trafic** (*traffic class*) **sau tipul de serviciu** (ToS) **din antetul IP al pachetelor trimise prin *socket-ul* curent**.

Deoarece implementarea nivelului retea poate ignora valoarea parametrului `tc` (in cazul in care **nu exista sau nu este activat controlul de trafic pentru servicii diferentiate**), **aplicatia trebuie sa interpreteze apelul ca pe o sugestie** (*hint*) data nivelului inferior.

Pentru protocolul IPv4 valoarea parametrului **`tc`** este interpretata drept **campurile precedenta si ToS**. **Campul ToS este creat ca set de biti** obtinut prin **aplicarea functiei logice OR valorilor**:

`IPTOS_LOWCOST = 0x02`	- indicand cerinte de **cost redus** din partea aplicatiei
`IPTOS_RELIABILITY = 0x04`	- indicand cerinte de **fiabilitate** din partea aplicatiei
`IPTOS_THROUGHPUT = 0x08`	- indicand cerinte de **banda larga** din partea aplicatiei
`IPTOS_LOWDELAY = 0x10`	- indicand cerinte de intarziere redusa (**timp real**) ale aplicatiei

Cel mai putin semnificativ bit al octetului **`tc`** e ignorat, el **trebuind sa fie zero** (*must be zero* – MZB).

Stabilirea bitilor in campul de precedenta (cei mai semnificativi 3 biti ai octetului **`tc`**) poate produce o **`SocketException`**, indicand imposibilitatea realizarii operatiei.

Pentru protocolul IPv6 valoarea parametrului **`tc`** este plasata in **campul `sin6_flowinfo`** al antetului IP.

5.4.4. Programe ilustrative pentru lucrul cu *socket*-uri datagrame (UDP)

Urmatorul program utilizeaza crearea de ***socket* UDP** pentru a **identifica porturile locale pe care exista conexiuni** (**pe care nu pot fi create servere**).

```
 1  import java.net.*;
 2  public class ScannerPorturiUDPLocale {
 3    public static void main (String args[]) {
 4      DatagramSocket serverUDP;
 5
 6      for (int portUDP=1; portUDP < 65536; portUDP++) {
 7        try {
 8          // Urmatoarele linii vor genera exceptie prinsa de blocul catch
 9          // in cazul in care e deja un server pe portul portUDP
10          serverUDP = new DatagramSocket(portUDP);
11          serverUDP.close();
12        }
13        catch (SocketException ex) {
14          System.err.println("Exista un server UDP pe portul " + portUDP);
15        }
16      }
17    }
18  }
```

Programul `ClientEcouDatagrameRepetitiv`, (**client pentru un server ecou UDP care permite trimiterea mai multor mesaje,** mesajul format dintr-un punct (`"."`) semnaland serverului terminarea mesajelor de trimis, clientul urmand sa isi termine executia):

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class ClientEcouDatagrameRepetitiv {
 4    public static void main (String args[]) throws IOException {
 5      BufferedReader inConsola  = new BufferedReader(new
 6                                      InputStreamReader(System.in));
 7      System.out.print("Introduceti adresa IP a serverului: ");
 8      String adresaServer = inConsola.readLine();
 9      System.out.print("Introduceti numarul de port al serverului: ");
10      int portServer = Integer.parseInt(inConsola.readLine());
11
12      byte[] bufferDate = new byte[1024];
13
14      DatagramPacket pachetDeTrimis;
15      DatagramPacket pachetPrimit = new DatagramPacket(bufferDate, 1024);
16
17      // Crearea socketului
18      DatagramSocket socketUDP = new DatagramSocket();
19      System.out.println("Client pentru serverul " + adresaServer + ":" +
20                         portServer + " lansat...");
21
22      // Utilizare socket client
23      while(true) {
24        // Citirea unei linii de la consola de intrare
25        System.out.print("Mesaj de trimis: ");
26        String mesajDeTrimis = inConsola.readLine();
27
28        // Plasarea datelor mesajului de trimis in buffer
29        bufferDate = mesajDeTrimis.getBytes();
30
31        // Constructia pachetului UDP de trimis
32        pachetDeTrimis=new DatagramPacket(bufferDate, mesajDeTrimis.length(),
33               InetAddress.getByName(adresaServer), portServer);
34
35        // Trimiterea pachetului UDP
36        socketUDP.send(pachetDeTrimis);
37
38        // Receptia unui pachet UDP - blocare in asteptare
39        socketUDP.receive(pachetPrimit);
40
41        // Obtinerea datelor pachetului ca tablou de octeti
42        bufferDate = pachetPrimit.getData();
43
44        // Reconstructia mesajului text
45        String mesajPrimit =
46                   new String(bufferDate, 0, pachetPrimit.getLength());
47
48        // Afisarea informatiilor primite
49        System.out.println("Mesaj primit de la " +
50                 pachetPrimit.getAddress().getHostName() + ":" +
51                 pachetPrimit.getPort() + ": \n" + mesajPrimit);
52
53        // Testarea conditiei de oprire
54        if (mesajPrimit.equals(".")) break;
55      }
56      // Inchiderea socketului
57      socketUDP.close();
58    }
59  }
```

Programul `ServerEcouDatagrameRepetitiv`, (**server care permite receptia si trimiterea in ecou a mai multor mesaje sosite de la un singur client**, mesajul format dintr-un punct (".") semnaland serverului terminarea mesajelor de trimis, serverul urmand sa isi termine executia):

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class ServerEcouDatagrameRepetitiv {
 4    public static void main (String args[]) throws IOException {
 5      BufferedReader inConsola  = new BufferedReader(new
 6                                      InputStreamReader(System.in));
 7      System.out.print("Introduceti numarul de port al serverului: ");
 8      int portServer = Integer.parseInt(inConsola.readLine());
 9
10      // Crearea socketului
11      DatagramSocket socketUDP = new DatagramSocket(portServer);
12      System.out.println("Server in asteptare pe portul "+portServer+ "...");
13
14      byte[] bufferDate = new byte[1024];
15
16      DatagramPacket pachetDeTrimis;
17      DatagramPacket pachetPrimit = new DatagramPacket(bufferDate, 1024);
18
19      // Utilizare socket server
20      while(true) {
21        // Receptia unui pachet UDP - blocare in asteptare
22        socketUDP.receive(pachetPrimit);
23
24        // Obtinerea datelor pachetului ca tablou de octeti
25        bufferDate = pachetPrimit.getData();
26
27        // Reconstructia mesajului text
28        String mesajPrimit =
29              new String(bufferDate, 0, pachetPrimit.getLength());
30
31        // Afisarea informatiilor primite
32        System.out.println("Mesaj primit de la " +
33                 pachetPrimit.getAddress().getHostName() + ":" +
34                 pachetPrimit.getPort() + ": \n" + mesajPrimit);
35
36        // Constructia pachetului UDP de trimis
37        pachetDeTrimis = new DatagramPacket(pachetPrimit.getData(),
38                                 pachetPrimit.getLength(),
39                                 pachetPrimit.getAddress(),
40                                 pachetPrimit.getPort());
41
42        // Trimiterea pachetului UDP
43        socketUDP.send(pachetDeTrimis);
44
45        // Testarea conditiei de oprire a servirii
46        if (mesajPrimit.equals(".")) break;
47      }
48      // Inchidere socket
49      socketUDP.close();
50    }
51  }
```

Pentru constructia datagramelor complexe pot fi utilizate **fluxurile `ByteArrayOutputStream` si `DataOutputStream`.**

Urmatoarea clasa contine metode utile crearii pachetelor UDP folosind fluxurile `ByteArrayOutputStream` si `DataOutputStream`.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5   *  Metode utile lucrului cu pachete UDP (datagrame) de trimis
 6   */
 7  public class UtilePachetTrimitereUDP {
 8
 9    ByteArrayOutputStream outTablou;
10    DataOutputStream      outDateFormatate;
11    byte[]                bufferDate;
12
13    /**
14     *  Construieste pachet UDP de trimitere cu continut text
15     */
16    public static DatagramPacket constructiePachetString(String text,
17                         String adresa, int port) throws IOException {
18      return new DatagramPacket(text.getBytes(), text.length(),
19                                InetAddress.getByName(adresa), port);
20    }
21
22    /**
23     *  Incepe constructia unui pachet UDP de trimitere complex
24     */
25    public DataOutputStream obtinereFluxDateFormatate() throws IOException{
26      outTablou = new ByteArrayOutputStream();
27      outDateFormatate = new DataOutputStream(outTablou);
28      return outDateFormatate;
29    }
30
31    /**
32     *  Incheie constructia unui pachet UDP de trimitere complex
33     */
34    public DatagramPacket incheiereConstructiePachet(String adresa,
35                                                     int port) throws IOException {
36      outDateFormatate.flush();
37      bufferDate = outTablou.toByteArray();
38      return new DatagramPacket(bufferDate, bufferDate.length,
39                                InetAddress.getByName(adresa), port);
40    }
41  }
```

Pentru extragerea datelor din datagramele complexe pot fi utilizate **fluxurile fluxurile `ByteArrayInputStream` si `DataInputStream`.**

Urmatoarea clasa contine metode utile extragerii datelor din pachetele UDP folosind fluxurile `ByteArrayInputStream` si `DataInputStream`.

```
 1  Import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Metode utile lucrului cu pachete UDP (datagrame) de primit
 5   */
 6  public class UtilePachetPrimireUDP {
 7
 8    DatagramPacket       datagrama;
 9    ByteArrayInputStream inTablou;
10    DataInputStream      inDateFormatate;
11    byte[]               bufferDate;
12
13    /**
14     *  Obtine String din pachet UDP cu continut text
15     */
16    public static String extrageString(DatagramPacket datagrama)
17                                                    throws IOException {
18      return new String(datagrama.getData(), 0, datagrama.getLength());
19    }
20
21    /**
22     *  Pregateste extragerea dintr-un pachet UDP complex
23     */
24    public DataInputStream obtineFluxDateFormatate(DatagramPacket datagrama)
25                                                    throws IOException {
26      inTablou = new ByteArrayInputStream(datagrama.getData(),
27                       datagrama.getOffset(), datagrama.getLength());
28      inDateFormatate = new DataInputStream(inTablou);
29      return inDateFormatate;
30    }
31  }
```

Urmatorul program client (`TestUtilePachetTrimitereUDP`) ilustreaza modul de utilizare al metodelor din clasele utilitare anterioare.

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Client test metode utile lucrului cu pachete UDP (datagrame) de trimis
 5   */
 6  public class TestUtilePachetTrimitereUDP {
 7
 8    /**
 9     *  Testarea/exemplificarea utilizarii clasei UtilePachetTrimitereUDP
10     */
11    public static void main (String args[]) throws IOException {
12      BufferedReader inConsola  = new BufferedReader(new
13                                      InputStreamReader(System.in));
14      System.out.print("Introduceti adresa IP a serverului: ");
15      String adresaServer = inConsola.readLine();
16      System.out.print("Introduceti numarul de port al serverului: ");
17      int portServer = Integer.parseInt(inConsola.readLine());
18
19      DatagramSocket socketUDP = new DatagramSocket();
20      System.out.println("Client pentru serverul " + adresaServer + ":" +
21                         portServer + " lansat...");
22
23      byte[] bufferDate = new byte[1024];
24
25      DatagramPacket pachetDeTrimis;
26
27      System.out.print("Mesaj de trimis: ");
28      String mesajDeTrimis = inConsola.readLine();
29
30      pachetDeTrimis = UtilePachetTrimitereUDP.constructiePachetString(
31                                  mesajDeTrimis, adresaServer, portServer);
32
33      InfoPachetUDP.afisareInfoPachetUDP(pachetDeTrimis, true);
34
35      socketUDP.send(pachetDeTrimis);    // Trimiterea pachetului UDP
36
37      UtilePachetTrimitereUDP utileTrimitere = new UtilePachetTrimitereUDP();
38
39      DataOutputStream flux = utileTrimitere.obtinereFluxDateFormatate();
40
41      byte octet = 100;
42      flux.writeByte(octet);
43
44      short intregScurt = 1000;
45      flux.writeShort(intregScurt);
46
47      flux.writeInt(100000);
48
49      flux.writeLong(1000000L);
50
51      flux.writeUTF(mesajDeTrimis + " bis");
52
53      pachetDeTrimis= utileTrimitere.incheiereConstructiePachet(adresaServer,
54                                                                portServer);
55      InfoPachetUDP.afisareInfoPachetUDP(pachetDeTrimis, true);
56
57      socketUDP.send(pachetDeTrimis);    // Trimiterea pachetului UDP
58
59      socketUDP.close();
60    }
61  }
```

Urmatorul program server (`TestUtilePachetPrimireUDP`) ilustreaza modul de utilizare al metodelor din clasele utilitare anterioare.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5   *  Server test metode utile lucrului cu pachete UDP (datagrame) de primit
 6   */
 7  public class TestUtilePachetPrimireUDP {
 8
 9    /**
10     *  Testarea/exemplificarea utilizarii clasei UtilePachetPrimireUDP
11     */
12    public static void main (String args[]) throws IOException {
13      BufferedReader inConsola  = new BufferedReader(new
14                                      InputStreamReader(System.in));
15
16      System.out.print("Introduceti numarul de port al serverului: ");
17      int portServer = Integer.parseInt(inConsola.readLine());
18
19      DatagramSocket socketUDP = new DatagramSocket(portServer);
20      System.out.println("Server in asteptare pe portul "+portServer+"...");
21
22      byte[] bufferDate = new byte[1024];
23
24      DatagramPacket pachetPrimit = new DatagramPacket(bufferDate, 1024);
25
26      // Receptia unui pachet UDP - blocare in asteptare
27      socketUDP.receive(pachetPrimit);
28
29      InfoPachetUDP.afisareInfoPachetUDP(pachetPrimit, false);
30
31      String text = UtilePachetPrimireUDP.extrageString(pachetPrimit);
32      System.out.println("Text primit: " + text);
33
34      // Receptia unui pachet UDP - blocare in asteptare
35      socketUDP.receive(pachetPrimit);
36
37      InfoPachetUDP.afisareInfoPachetUDP(pachetPrimit, false);
38
39      UtilePachetPrimireUDP utilePrimire = new UtilePachetPrimireUDP();
40
41      DataInputStream flux =
42            utilePrimire.obtineFluxDateFormatate(pachetPrimit);
43
44      byte octet = flux.readByte();
45      System.out.println("Octet primit: " + octet);
46
47      short intregScurt = flux.readShort();
48      System.out.println("Intreg scurt primit: " + intregScurt);
49
50      int intreg = flux.readInt();
51      System.out.println("Intreg primit: " + intreg);
52
53      long intregLung = flux.readLong();
54      System.out.println("Intreg lung primit: " + intregLung);
55
56      String text2 = flux.readUTF();
57      System.out.println("Text primit: " + text2);
58
59      socketUDP.close();
60    }
61  }
```