Aplicații integrate pentru întreprinderi

Laborator 6

30.11.2010

Comunicația interproces folosind socketi TCP în Java

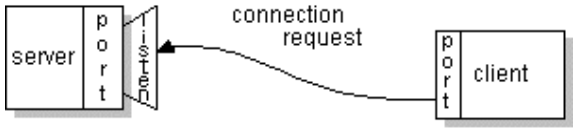
Scopul laboratorului îl reprezintă utilizarea mecanismului oferit de socketii TCP în limbajul de programare Java în scopul realizării comunicației între diferite procese care rulează la nivelul sistemului de operare.

1. Ce sunt socketii ?

Socketii sunt metode de comunicație de nivel scăzut între procese întrucât procesele comunicante pot rula pe maşini cu arhitectură sau sistem de operare diferite. Canalul de comunicație între procese este *bidirecțional*, dar *nu se ocupă de structura datelor transmise*.

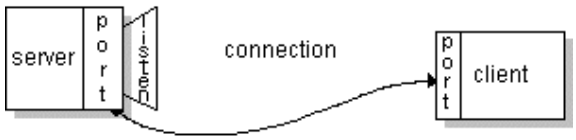
Structura aplicațiilor care folosesc socketi TCP este de regulă client-server.

Serverul crează un socket (căruia îi asociază un port[1]) pentru ca ulterior să asculte cereri (de conexiune) de la clienți.



În condițiile în care cererea de conexiune de la client a fost acceptată, serverul crează un nou socket[2] pe care îl asociază cu acelaşi port (pe care ascultă), celălalt capăt fiind reprezentat de adresa IP şi portul clientului.

Clientul va crea un socket[3] (folosind adresa IP şi portul[4] unde ascultă socket-ul de pe server) după care va putea transmite pachete în şi dinspre server, în cazul în care conexiunea este acceptată.



[1] Nu este corect să se facă asocierea (logică) între un socket şi o maşină (fizică sau virtuală) ci între un socket şi o interfață de rețea din cadrul maşinii.

[2] Un nou socket este necesar pentru a putea să continue să asculte în continuare şi alte noi cereri (de conexiune) de la clienți pe socket-ul inițial, comunicând şi cu clientul.

[3] Sistemul de operare alocă în mod automat un port pe care comunică clientul.

[4] Clientul cunoaşte adresa IP a maşinii unde rulează serverul precum şi portul pe care ascultă socket-ul.

Aşadar, serverul şi clientul stabilesc o conexiune unul la celălalt, fiecare dintre componente creând un socket la capătul propriu al conexiunii, în care pot scrie şi citi informații.

Un socket poate fi definit ca fiind un capăt al unei conexiuni bidirecționale între două aplicații care rulează într-o rețea de calculatoare.

Un socket are un port asociat, astfel încât protocolul TCP să poată identifica aplicația spre care transmite datele.

Un capăt al conexiunii este definit printr-o adresă IP şi un port.

O conexiune TCP este identificată în mod ***complet*** prin cele două capete ale conexiunii[5].

Comunicația dintre server şi client folosind socketi TCP este sigură, deoarece se asigură faptul că nu se pierd date, acestea fiind primite în aceeaşi ordine în care au fost transmise[6].

2. Ce metode pune la dispoziție API-ul Java pentru socketi TCP ?

În limbajul de programare Java, pachetul `java.net` pune la dispoziția programatorilor două clase: `ServerSocket` şi `Socket` care implementează partea de conexiune de la nivelul serverului, respectiv a clientului.

- clasa `java.net.ServerSocket` implementează un tip de socket care poate fi folosit de un server pentru a accepta conexiuni de la clienți.
- clasa `java.net.Socket` reprezintă un capăt al conexiunii bidirecționale între cele două programe care comunică prin rețeaua de calculatoare; implementarea este *independentă de platformă*, făcând transparente detaliile specifice platformei pe care rulează aplicația pentru programator.

Clasa `ServerSocket`

Clasa implementează socketi de tip server care aşteaptă cereri provenite din rețea. Sunt realizate prelucrări pe baza interogării şi (eventual) se furnizează şi rezultate.

Operațiile clasei `ServerSocket` sunt realizate de o instanță a clasei `SocketImpl`. Aplicația poate schimba „fabrica" de socketi ce crează implementarea pentru a crea socketi corespunzători.

`ServerSocket ([port, [lungime_coada_asteptare, adresa]]])`	Fără parametri, crează un socket care nu este legat la nici un port. Dacă portul specificat este zero, este alocat automat un port (liber) de către sistemul de operare. Parametrul lungime_coada_asteptare reprezintă un întreg pozitiv care specifică numărul de conexiuni care se poate afla simultan în coada de aşteptare pentru a fi acceptate. Adresa este folosită pentru maşinile cu mai multe interfețe de rețea, pentru a specifica adresa pe care va asculta socket-ul.

[5] Proprietatea permite conexiuni multiple între client şi server.

[6] O astfel de caracteristică este garantată de protocolul TCP pe care se bazează mecanismul folosind socketi, protocol de comunicare punct la punct sigur.

`bind (adresa_port, [lungime_coada_asteptare])`	Leagă socket-ul de o adresă IP şi un port (obiect de tip `SocketAddress`).
`getInetAddress()`	Întoarce adresa locală a socket-ului de tip server.
`getLocalPort()`	Întoarce portul pe care ascultă socket-ul de tip server
`getLocalSocketAddress()`	Întoarce adresa capătului de conexiune la care este legat sau `null` dacă nu este legat încă. Rezultatul este un obiect de tip `SocketAddress`.
`accept()`	Aşteaptă o cerere (de conexiune) pe care o acceptă. Metoda este *blocantă* până când este realizată conexiunea. Înainte de a se crea socket-ul, este apelata metoda `checkAccept` din modulul de gestiune al securității.
`implAccept(socket)`	Metodă care suprascrie metoda `accept` din subclasele clasei `ServerSocket`.
`close()`	Închide socket-ul şi canalul asociat, daca există. Firele de execuție care aşteaptă în metoda `accept` fiind blocate vor arunca excepția `SocketException`.
`getChannel()`	Întoarce obiectul `ServerSocketChannel` asociat socket-ul (dacă acesta a fost creat prin metoda `ServerSocketChannel.open`).
`isBound()`	Precizează starea de legare a socket-ului.
`isClose()`	Precizează starea de închidere a socket-ului.
`setSoTimeout(timp_expirare)`	Stabileşte valoarea variabilei SO_TIMEOUT (măsurată în milisecunde) reprezentând timpul după expirarea căruia metoda `accept` (înaintea căreia trebuie apelată) aruncă excepția `SocketTimeoutException`.[7]
`getSoTimeout()`	Întoarce valoarea variabilei SO_TIMEOUT[8].
`setReuseAddress([da|nu])`	Permite socket-ului de tip server să fie legat prin metoda `bind` la o adresă (variabila de tip `SocketAddress`) după ce conexiunea respectiva a fost închisă găsindu-se într-o stare nedefinită pentru o perioadă de timp (`TIME_WAIT`)[9]. Opțiunea este reținută prin intermediul variabilei SO_REUSEADDR.
`getReuseAddress()`	Verifică dacă variabila SO_REUSEADDR este activată sau dezactivată.
`setSocketFactory(fabrica_socket)`	Stabileşte fabrica de implementare a socket-ilor pentru aplicație[10]. Crearea serverului de tip server foloseşte metoda `createSocketImpl` pentru a obține implementarea socket-ului. Metoda `checkSetFactory` verifică dacă operația este permisă.

[7] Socket-ul de tip server continuă să funcționeze şi după aruncarea acestei excepții.
[8] O valoare 0 indică faptul că opțiunea este dezactivată.
[9] Comportamentul metodei nu mai este definit din momentul în care socket-ul este legat.
[10] Specificarea poate fi realizată o singură dată.

`setReceiveBufferSize(lungime)`	Stabileşte valoarea opțiunii SO_RCVBUF după ce socket-ul este acceptat, reprezentând dimensiunea zonei tampon[11] (interne) folosită de socket şi dimensiunea ferestrei TCP care este anunțată la celălalt capăt al conexiunii.
`getReceiveBufferSize()`	Obține valoarea opțiunii SO_RCVBUF care va fi folosită de socketii acceptați.
`setPerformanceParameters (int timp_conexiune, int latenta, int latime_de_banda)`	Stabileşte preferințele de performanță[12] pentru socket-ul de tip server. Parametrii specifică importanța relativă a timpului de conexiune scurt, latenței mici şi lățimii de bandă ridicate. Valorile sunt comparate între ele, numerele mai mari semnifică o preferință pentru parametrul în cauză.

Clasa `Socket`

Clasa implementează socketi de tip client (numiți mai simplu socketi), reprezentând un capăt al conexiunii dintre două maşini.

`Socket ([adresa, port, [flux]])`
`Socket (adresa, port, adresa_locala, port_local)`
`Socket (proxy)`
`Socket (implementare_socket)`
`Socket (nume_masina, port, [flux])`
`Socket (nume_masina, port, adresa_locala, port_local)`
	Fără parametri, crează un socket neconectat, având tipul implicit `SocketImpl`. Se pot specifica diferite tipuri de proxy. Se poate indica o implementare pentru socket definită de utilizator.
	Socket-ul poate fi conectat la o maşină specificată prin nume[13]/adresă (IP) şi prin port[14]. Este posibil ca simultan conexiunii la server (la distanță), socket-ul sa fie legat la adresa locală şi portul local.
	Parametrul flux specifica tipul de socket (de tip flux sau datagramă[15]).

`connect (adresa_port, [timp_expirare])`	Conectează socket-ul la server specificat printr-o variabilă de tip `SocketAddress`, specificând totodată şi un timp de expirare.
`bind (adresa_port)`	Leagă socket-ul la adresa locală[16].
`getInetAddress()`	Întoarce adresa la care este conectat serverul.
`getLocalAddress()`	Întoarce adresa spre care socketul este îndreptat.
`getPort()`	Întoarce portul (la distanță) la care este conectat socket-ul.

[11] Daca se doresc valori mai mari de 64K, specificarea trebuie să se facă înaintea legării la adresă (se apelează constructorul fără parametri, se apeleaza metoda `setReceiveBufferSize()` şi apoi legarea se face prin `bind()`).

[12] Apelarea metodei după ce socket-ul de tip server a fost legat nu are nici un efect.

[13] În cazul în care numele este `null`, se va încerca conexiunea la interfața de loopback.

[14] Dacă aplicația a specificat o fabrică de socketi, este apelata metoda createSocketImpl în scopul de a descrie implementarea propriu-zisă a socket-ului.

[15] Metoda este **deprecated**, în mod curent se folosesc în mod distinct clasele `Socket` şi `DatagramSocket`.

[16] Dacă adresa este `null`, sistemul de operare va alege un port la întâmplare şi o adresă locală pentru a lega socket-ul.

`getLocalPort()`	Întoarce portul local la care socket-ul este legat.
`getRemoteSocketAddress()`	Întoarce adresa capătului conexiunii la care socket-ul este conectat sau `null`, dacă nu este conectat. Rezultatul este furnizat sub forma unui obiect având tipul `SocketAddress`.
`getLocalSocketAddress()`	Întoarce adresa capătului conexiunii de care socket-ul este legat, sau `null`, dacă nu este legat (încă).
`getChannel()`	Întoarce obiectul de tipul `SocketChannel` (unic) asociat cu socket-ul, în condițiile în care acesta există[17].
`getInputStream()`	Oferă fluxul de intrare pentru socket. Dacă socket-ul are un canal asociat, atunci fluxul de intrare va delega toate operațiile canalului[18]. Conexiunea poate fi coruptă de maşina la distanță sau de aplicația de rețea (spre exemplu, pentru conexiunile TCP, o reinițializare de conexiune). • aplicația de rețea poate ignora informația care se găseşte în zona de memorie tampon folosită de socket; datele care nu sunt ignorate sunt disponibile prin funcția `read`; • dacă nu există octeți în zona de memorie tampon sau toate informațiile au fost preluate deja de `read`, apelurile ulterioare ale metodei read vor arunca excepția `IOException`; • dacă nu se mai găsesc date pe socket şi acesta nu a fost închis, metoda `available` asociată fluxului de intrare întoarce valoarea 0. Închiderea obiectului de tip `InputStream` întors determină închiderea socket-ului asociat.
`getOutputStream()`	Oferă fluxul de ieşire pentru socket. Dacă socket-ul are un canal asociat, atunci fluxul de ieşire va delega toate operațiile canalului[19]. Închiderea obiectului de tip `OutputStream` întors determină închiderea socket-ului asociat.
`setTcpNoDelay([da|nu])`	Activează sau dezactivează variabila TCP_NODELAY (dezactiveaza sau activează algoritmul lui Nagle[20]).
`getTcpNoDelay()`	Verifică dacă variabila TCP_NODELAY este activată sau nu.

[17] Un socket are un obiect de tip `SocketChannel` asociat dacă şi numai dacă acesta a fost creat prin metodele `SocketChannel.open` sau `SocketChannel.accepts`.

[18] În situația în care canalul nu este blocant, operațiile de intrare de tip `read` vor provoca excepția `IllegalBlockingModeException`.

[19] În situația în care canalul nu este blocant, operațiile de intrare de tip `write` vor provoca excepția `IllegalBlockingModeException`.

[20] RFC 896: *„Congestion Control in IP/TCP Internetworks"* – descrie modalitatea în care poate fi îmbunătățită eficiența rețelelor TCP/IP prin reducerea numărului de pachete care trebuie transmise prin intermediul rețelei.

`setSoLinger([da|nu], intarziere)`	Activează sau dezactivează variabila SO_LINGER specificând timpul de întârziere în secunde, valoarea de întârziere maximă fiind dependentă de platformă şi afectând doar operația de închidere a socket-ului.
`getSoLinger()`	Întoarce valoarea variabilei SO_LINGER, -1 indicând faptul că opțiunea este dezactivată.
`sendUrgentData(data)`	Trimite un octet de date urgente la socket, ultimii opt biți din parametrul `data`. Octetul „urgent" este trimis după scrieri anterioare în obiectul `OutputStream` asociat socketului şi înainte de alte scrieri ulterioare.
`setOOBInline([da|nu])`	Activează sau dezactivează variabila OOBINLINE (transmiterea de date urgente TCP). Implicit, opțiunea estge dezactivată şi datele urgente TCP transmise pe un socket sunt respinse. Opțiunea trebuie activată în condițiile în care este necesar lucrul cu date urgente. Datele urgente sunt primite „în rând" cu datele normale şi nu se poate distinge între cele două categorii, decât dacă acest lucru este realizat de un protocol de nivel mai înalt.
`getOOBInline()`	Verifică dacă variabila OOBINLINE este activată sau nu.
`setSoTimeout(timp_expirare)`	Stabileşte valoarea variabilei SO_TIMEOUT (măsurată în milisecunde) reprezentând intervalul de timp în care metoda `read` (înaintea căreia trebuie apelată) va rămâne blocată, după care aruncă excepția `SocketTimeoutException`.
`getSoTimeout()`	Întoarce valoarea variabilei SO_TIMEOUT.
`setSendBufferSize(lungime)`	Stabileşte valoarea variabilei SO_SNDBUF la valoarea specificata prin parametrul `lungime`, fiind folosită de aplicațiile de rețea ca un indiciu pentru dimensiunea zonelor de memorie tampon destinate operațiilor de intrare/ieşire de la nivelul rețelei.
`getSendBufferSize()`	Obține valoarea variabilei SO_SNDBUF care este dimensiunea zonei de memorie tampon care va fi folosită ca ieşire pentru socket.
`setReceiveBufferSize(lungime)`	Stabileşte valoarea variabilei SO_RCVBUF la valoarea specificata prin parametrul `lungime`, fiind folosită de aplicațiile de rețea ca un indiciu pentru dimensiunea zonelor de memorie tampon destinate operațiilor de intrare/ieşire de la nivelul rețelei[21]. Valoarea este folosită pentru stabilirea dimensiunii ferestrei TCP care este transmisă la celălalat capăt al conexiunii, putând fi modificată în orice moment.

[21] Creşterea zonei de memorie pentru recepționarea mesajelor poate îmbunătăți performanța operațiilor de intrare/ieşire de la novelul rețelei pentru conexiuni de volum mare, reducerea zonei de memorie conducând la micşorarea cozii de aşteptare a datelor.

`getReceiveBufferSize()`	Obține valoarea variabilei SO_RCVBUF care este dimensiunea zonei de memorie tampon care va fi folosită ca intrare pentru socket.
`setKeepAlive([da|nu])`	Stabileşte valoarea variabilei SO_KEEPALIVE.
`getKeepAlive()`	Verifică dacă variabila SO_KEEPALIVE este activată sau nu.
`setTrafficClass(clasa_trafic)`	Stabileşte clasa de trafic sau octetul type-of-service din antetul IP pentru pachetele trimise de la socket. Unele implementări pot ignora acest câmp, aplicațiile considerându-l doar un indiciu. Valorile pentru parametrul `clasa_trafic` se înscriu în intervalul `[0,255]`[22].
`getTrafficClass()`	Obține clasa de trafic sau octetul type-of-service din antetul IP pentru pachetele trimise de la socket. Deoarece unele implementări ignoră valoarea stabilită prin metoda `setTrafficClass`, poate fi întoarsă o valoare diferită decât cea specificată.
`setReuseAddress([da|nu])`	Permite socket-ului (de tip client) să fie legat prin metoda `bind` la o adresă (variabila de tip `SocketAddress`) după ce conexiunea respectiva a fost închisă găsindu-se într-o stare nedefinită pentru o perioadă de timp (`TIME_WAIT`). Opțiunea este reținută prin intermediul variabilei SO_REUSEADDR.
`getReuseAddress()`	Verifică dacă variabila SO_REUSEADDR este activată sau dezactivată.
`close()`	Închide socket-ul. Firele de execuție care sunt blocate în operații de intrare/ieşire vor arunca excepția `SocketException`. După ce un socket a fost închis, nu mai este disponibil pentru alte operații din rețea (nu poate fi reconectat sau relegat), trebuind să fie creat un nou socket.
`shutDownInput()`	Plasează fluxul de intrare pentru socket la sfârşit. Datele transmise spre fluxul de intrare al socket-ului sunt ignorate după ce se trimite pentru ele confirmare de primire. Dacă se citeşte dintr-un flux de intrare după apelul metodei, se va întoarce EOF.

[22] Valorile sunt specificate în RFC 1349 (IPv4) fiind un octet cu precedență (bitul cel mai puțin semnificativ este ignorat şi este întotdeauna 0 – MBZ = must be zero bit), construit cu operații OR (pe biți):

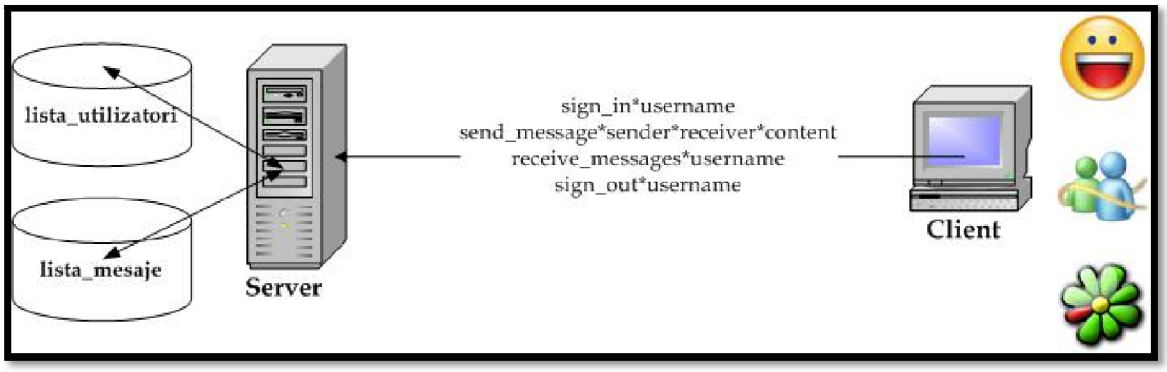
- `IPTOS_LOWCOST (0x02)`
- `IPTOS_RELIABILITY (0x04)`
- `IPTOS_THROUGHPUT (0x08)`
- `IPTOS_LOWDELAY(0x10)`

Operațiile nepermise genereaza o excepție SocketExceptuin.
Câmpul poate fi modificat pe toata perioada existenței conexiunii, dar depinde de implementare dacă operația are efect sau nu.
Pentru IPv6, valoarea parametrului se găseşte în câmpul sin6_flowinfo al antetului IP.

`shutDownOutput()`	Dezactivează fluxul de ieşire pentru socket. Pentru un socket TCP, orice date scrise anterior vor fi trimise urmate de secvența de încheiere a conexiunii prevăzută de protocolul TCP. Dacă se încearcă scrierea într-un flux de ieşire după apelul metodei, va fi generată o excepție `IOException`.
`toString()`	Converteşte socket-ul la un şir de caractere.
`isConnected()`	Precizează starea de conectare a socket-ului.
`isBound()`	Precizează starea de legare a socket-ului.
`isClose()`	Precizează starea de închidere a socket-ului.
`isInputShutDown()`	Precizează dacă partea de citire a socket-ului este închisă.
`isOutputShutDown()`	Precizează dacă partea de scriere a socket-ului este închisă.
`setSocketFactory(fabrica_socket)`	Stabileşte fabrica de implementare a socketț-ilor pentru aplicație. Crearea serverului de tip server foloseşte metoda `createSocketImpl` pentru a obține implementarea socket-ului. Metoda `checkSetFactory` verifică dacă operația este permisă.
`setPerformanceParameters (int timp_conexiune, int latenta, int latime_de_banda)`	Stabileşte preferințele de performanță pentru socket-ul de tip server. Parametrii specifică importanța relativă a timpului de conexiune scurt, latenței mici şi lățimii de bandă ridicate. Valorile sunt comparate între ele, numerele mai mari semnifică o preferință pentru parametrul în cauză.

În cele ce urmează, comunicarea în cadrul unei aplicații distribuite folosind socketi TCP va fi ilustrată pe cazul particular al unui program de tipul mesagerie instantă.

Serverul va accepta mai multe conexiuni de la clienți şi va reține într-un obiect intern lista utilizatorilor care sunt conectați la un moment dat precum şi lista mesajelor care nu au fost încă transmise de la sursă la destinație.



Mesajele care vor putea fi interpretate de server sunt următoarele:

`sign_in*username`	Utilizatorul identificat prin şirul de caractere `username` se va adăuga de server în lista utilizatorilor conectați dacă un utilizator cu acelaşi nume nu există deja în lista de utilizatori.
`send_message*sender*receiver*content`	Mesajul având conținutul `content` este trimis de la utilizatorul identificat prin şirul de caractere `sender` către utilizatorul identificat prin şirul de caractere `receiver`.
`receive_messages*username`	Server-ul va transmite un şir de caractere conținând mesajele destinate utilizatorului identificat prin şirul de caractere `username` (conținând destinatarul şi conținutul), totodată din lista de mesaje.
`sign_out*username`	Utilizatorul identificat prin şirul de caractere `username` se va elimina de server din lista utilizatorilor conectați (dacă utilizatorul cu acelaşi nume există deja în lista de utilizatori).

Formatul comenzilor corespunzătoare transmise de client este[23]:

1. `sign_in*username`[24]
2. `send_message*receiver*content`
3. `receive_messages`
4. `sign_out`

Conexiunea se va încheia atunci când clientul va specifica comanda `end_connection`.

Protocolul de comunicare între server şi client este unul de tip ping-pong (serverul sau clientul transmit un mesaj după care se blochează aşteptând mesajul de la celălalt capăt al socket-ului).

3. Programarea unui server

Serverul va crea un obiect de tip `ServerSocket` care va asculta pe un port (specificat ca parametru în linia de comandă). O excepție va fi generată dacă portul pe care se încearcă legarea socket-ului de tip server este deja ocupat.

```
try
{
    serverSocket = new ServerSocket(port);
}
catch (IOException e)
{
    System.err.println("Could not listen on port: "+port);
    System.exit(-1);
}
```

[23] Ordinea comenzilor este cea de mai jos, cu precizarea că instrucțiunile 2 şi 3 pot fi specificate în orice ordine.

[24] După specificarea şirului de caractere care identifică utilizatorul în procesul de înregistrare, nu mai este nevoie ca acest lucru să se facă explicit de catre utilizator, informația (către server) fiind adăugată în mod automat de aplicație la construirea mesajului.

`ServerSocket` este o clasă din pachetul `java.net` care oferă o implementare independentă de platformă pentru socketi de tip server.

Dacă legarea socket-ului de tip server la portul indicat s-a făcut cu succes, atunci el va putea accepta conexiuni de la client:

```
        while (listening)
            try
            {
                new ConnectionThread(serverSocket.accept()).start();
            }
            catch (Exception e)
            {
                System.err.println("Error while creating the socket.");
                System.exit(-1);
            }
```

Metoda `accept` se blochează până când un client cere o conexiune la adresa şi portul pe care a fost creat socket-ul de tip server (în cazul nostru, localhost şi portul specificat în linia de comandă). Când conexiunea a fost stabilită cu succes, metoda întoarce un obiect de tip `Socket` care este legat la aceeaşi adresă şi port ca şi socket-ul de tip server şi are ca adresă şi port la distanță pe cele ale clientului.

Serverul va putea asculta în continuare pe socket-ul de tip server (inițial) şi va putea comunica cu clientul pe socket-ul nou creat (întors ca rezultat al metodei `accept`). Pentru că metoda `accept` este blocantă, este necesar ca fiecare dintre aceste operație să fie realizată pe un fir de execuție separat, lansându-se pentru comunicarea cu clientul un nou fir de execuție.

```
public class ConnectionThread extends Thread
{
    @Override
    public void run()
    {
      try
        {
          PrintWriter out =
        new PrintWriter(socket.getOutputStream(), true);
          BufferedReader in =
        new BufferedReader(new InputStreamReader(socket.getInputStream()));

          String message;

          while ((message = in.readLine()) != null)
          {
              out.println(analyze(message));
              if (message.equals("close connection"))
                  break;
          }

          out.close();
          in.close();
          socket.close();
      }
      catch (IOException e) { e.printStackTrace(); }
    }
}
```

Comunicarea dintre server şi client se face obținând fluxurile de intrare şi de ieşire ale socket-ului şi creând obiecte de tip reader/writer (`BufferedReader`/`PrintWriter`) prin care se vor putea citi, respectiv scrie mesaje din şi în cadrul socket-ului.

Serverul va citi într-o buclă `while` mesajele de la client (`in.readLine()`), analizându-le în metoda `analyze(String message)`.

Condițiile în care comunicația între server şi client este menținută sunt:

- clientul transmite mesaje care nu sunt vide -> `in.readLine()!=null`;
- mesajul primit de la client nu este cel de încheiere al conexiunii

```
if (message.equals("close_connection"))
    break;
```

După prelucrările pe care le face, serverul trimite înapoi clientului rezultatele pe care le-a obținut după care se va bloca aşteptând următorul mesaj de la client.

La încheierea comunicației, sunt închise fluxurile de intrare şi de ieşire şi apoi socket-ul, eliberându-se resursele utilizate.

4. Programarea unui client

Clientul va crea un obiect de tip Socket care se va conecta la server, folosind adresa şi portul pe care ascultă socket-ul de tip server.

```
socket = new Socket(adresa, port);
```

Trebuie făcută distincția între *portul „la distanță”* specificat ca parametru de client la stabilirea conexiunii cu server-ul, specificând portul pe care ascultă[25] socket-ul de tip server şi *portul local*, valoare alocată de sistemul de operare automat şi care caracterizează portul pe care comunică socket-ul pe client.

Ca şi pe server, sunt obținute fluxurile de intrare şi de ieşire pentru socket, creându-se obiecte de tip reader/writer:

```
out = new PrintWriter(socket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

Comunicarea dintre client şi server se face într-o buclă, în care sunt primite comenzi de la utilizator (introduse de la tastatură) până când se specifică încheierea conexiunii printr-un şir de caractere predefinit `end_connection`.

```
do
{
    fromClient= keyboardInput.readLine();
    if (fromClient != null)
    {
        System.out.println("Client: " + fromClient);
        out.println(fromClient);
    }
```

[25] Socket-ul care este creat pe server în momentul conectării unui client comunică pe acelaşi port cu socket-ul de tip server care acceptă conexiuni de la clienți.

```
                fromServer = in.readLine();
                if (fromServer == null)
                    break;
                System.out.println("Server: " + fromServer);
                if (fromServer.equals("end_connection"))
                    break;
            }
            while (fromServer != null);
```

Mesajele citite de la tastatură sunt transmise server-ului, după care clientul se blochează (în metoda `readLine()`) aşteptând să primească răspunsul pentru informația trimisă.

La încheierea comunicației, sunt închise fluxurile de intrare şi de ieşire (specifice socket-ului şi mecanismului de citire de la tastatură) şi apoi socket-ul, eliberându-se resursele utilizate.

De remarcat faptul că acest algoritm funcționează după principiul:

```
transmite mesaj;
primeste mesaj;
```

şi că întotdeauna raportul între mesajele primite de la server şi transmise de la client este 1:1. Dacă se doreşte ca mesajele să poată fi transmise şi nepredictibil (deci nu ca răspuns la un mesaj), se vor crea doi socketi, unul pentru citire şi unul pentru scriere, lucrul cu fiecare dintre ele făcându-se într-un fir de execuție separat.

5. Mecanisme de sincronizare în Java

De multe ori, în aplicațiile distribuite, accesul la resurse comune se face din mai multe fire de execuție, existând posibilitatea ca atunci când acestea sunt accesate simultan de mai multe procese să se producă coruperea datelor.

Începând cu versiunea Java 1.5, a fost dezvoltat pachetul `java.util.concurrent` pentru lucrul în mediu distribuit care pune la dispoziția utilizatorilor colecții care pot fi folosite pentru accesul concurent precum şi mecanisme de sincronizare de tip mutex, barieră, semafor.

O clasă care oferă funcționalitate de tip mutex este `RentrantLock` din pachetul `java.util.concurent.lock`. Obiectul instanță al clasei `ReentrantLock` este deținut de ultimul fir de execuție care a reuşit să îl obțină fără a-l elibera. Constructorul clasei poate primi un parametru care să specifice echitatea acordării accesului la resursă (*eng.* fairness) prin intermediul căruia se stabileşte politica conform căreia firul de execuție care a formulat cererea de acces în urmă cu cel mai mult timp va primi accesul la resursă cel mai curând.

Este pusă la dispoziția utilizatorilor şi o metodă `tryLock` în două variante (cu şi fără parametri) care va încerca obținerea accesului la resursă în intervalul de timp specificat prin intermediul parametrilor. În cazul acestei metode, accesul la resursă nu va fi obținut conform politicii de corectitudine dacă în momentul cererii există alte fire de execuție care aşteaptă să obțină accesul la resursă.

```
lock.tryLock() = lock.tryLock(0, TimeUnit.SECONDS);
```

Este recomandat ca eliberarea accesului la resursă să se facă pe ramura `finally` a unui bloc `try...catch` astfel încât să se evite situația în care o resursă rămâne blocată datorită faptului că s-a generat o excepție undeva în cod.

```
private final ReentrantLock lock = new ReentrantLock();

...

lock.lock();
try
{
    ...
}
finally
{
    lock.unlock();
}
```

Activitate de laborator

(5p) 1. Completați metoda `public String analyze (String message)` din fişierul `ConnectionThread.java` astfel încât să trateze toate cazurile de mesaje care pot veni de la client.

(5p) 2. Completați metodele

- `public static void SignIn (String username)`
- `public static void SendMessage (String sender, String receiver, String content)`
- `public static String ReceiveMessages (String username)`
- `public static void SignOut (String username)`

din fişierul `MessengerServer.java` astfel incat să actualizeze corespunzător variabilele `users` şi `messages`.

(1p) 3. Modificați metodele `SignIn`, `SendMessage`, `ReceiveMessages`, `SignOut` astfel încât accesul la resursele users şi messages să se facă în mod sincronizat.

(2p) 4. Creați câte un socket şi câte un fir de execuție pentru fiecare din operațiile de citire, respectiv de scriere între client şi server.