**Programare concurenta in Java Mesagerie instant**

Cuprins:

1. Despre programare concurenta

       Fire de executie

-        Ce este un fir de executie?

-        Crearea firelor de executie

       Extinderea clasei Thread

       Interfata Runnable

-        Starile firelor de executie

       New

       Runnable

       Sleep

       Dead

-        Probleme de sincronizare

-        Problema Producator - Consumator

-        Realizarea excluderii reciproce

-        Wait, Notify, NotifyAll

-        Gruparea firelor de executie

       Java.util.concurent

-        Blocking Queue

-        Thread Pool Executor

1. Comunicarea prin socket-uri

-        Clasele Socket si ServerSocket

-        Comunicarea cu un singur client

-        Comunicarea cu mai multi clienti

-        Plain old sockets -  transmitere de fisiere

1. RMI (Remote Method Invocation)
2. Procesarea documentelor XML
3. Aplicatie

       Descriere

       Implementare

-        Server-client (comunicatios + conversation + avatarDownloading) (sockets)+ + ClientPersistence

-        threads (ThreadPoolExecutor)

-        messageType (XML)

-        Login (RMI)+ ServerPersistence

-        sending files (plain old sockets)

       Capturi

1.

**Programarea concurenta**este activitatea de construire a unui program continand procese multiple care se executa in paralel, ceea ce presupune o anumita cooperare intre ele.

Sunt necesare anumite mecanisme care sa asigure cumunicarea si sincronizarea intre procese.

Esenta programarii concurente nu depinde de numarul proceselor fizice disponibile.

Putem considera in permanenta ca dispunem de suficiente procesoare fizice pentru a atasa cate unul fiecaruia dintre procesele concurente.

De asemenea putem considera ca avem la dispozitie un singur procesor, care aloca proceselor intervale de timp aleatoare.

Putem considera ca este folosit modelul aleator, care consta in reperarea ciclica a urmatoarelor actiuni de carte fiecare procesor fizic:

* Alege aleator unul dintre procesele carora nu ii este alocat un procesor fizic
* Executa instructiunile sale, un interval aleator de timp.

Programele concurente pot sa aiba o comportare **non-deterministica**, adica pot produce rezultate diferite cand sunt rulate pe aceleasi date de intrare. Acest aspect nu este de dorit in cele mai multe cazuri, dar se datoreaza numarului mare de executii legale care difera prin ordinea de executie a operatiilor componente ale unui program.

Unele cazuri sunt favorabile programului altele nu. De aceea limbajele de programare concurenta  trebuie sa furnizeze un mod de a restrictiona executia programului astfel incat doar ca 343h74d zurile in care operatiile se executa intr-o ordine compatibila cu scopul urmarit sa aiba loc.

Limbajele fac acest lucru furnizand facilitati pentru **sincronizarea proceselor**.

Studiul acestora si modul in care pot fi folosite pentru a elimina toate cazurile nefavorabile sunt tema centrala de studiu al programarii concurente.

Programarea concurenta este menita sa rezolve interactiunile dintre procese. Atentia principala este indreptata asupra modalitatilor in care pot fi realizate interactiunile dorite si mai putin asupra a ceea ce realizeaza programele secventiale ce constituie fiecare proces in parte.

Sunt tratate in mod special probleme de simulare a unor situatii concurente reale si nu probleme care urmaresc obtinerea unui rezultat numeric.

Aceste procese sunt in competitie pentru accesarea resurselor critice si coopereaza pentru realizarea anumitor task-uri.

Avand data specificatia problemei care trebuie rezolvata decizia care trebuie luata este **cum trebuie ea impartita in procese**si cate anume sunt necesare si mai ales cum trebuie acestea sa interactioneze.

2.         Firele de executie fac trecerea de la programarea secventiala la programarea concureta.

Un program secvential reprezinta modelul clasic de program care are un inceput, o secventa de executie a instructiunilor sale si un sfarsit. Un program aflat in executie se numeste proces.

Un sistem de operare cum ar fi MS-DOS nu este capabil sa execute decat un singur proces la un moment dat, in timp ce un sistem de operare multitasking, cum ar fi Windows sau UNIX poate rula oricate procese in acelasi timp (concurent) folosind diverse strategii de alocare a procesorului pentru fiecare dintre ele.

Un fir de executie este similar unui proces secvential in sensul ca are un inceput, o secventa de executie si un sfarsit.

Diferenta dintre un fir de executie si un proces consta in faptul ca un fir de executie nu poate rula independent, ci trebuie sa ruleze in cadrul unui proces.

Un fir de executie trebuie vazut ca un flux de instructiuni care se executa in interiorul unui proces.

Un proces poate sa fie format din mai multe asemenea fire, care se executa in paralel, avand, insa in comun toate resursele principale caracteristice procesului.

Prin urmare, in interiorul unui proces, firele de executie sunt entitati care ruleaza in paralel, impartind intre ele zona de date si executand portiuni distincte din acelasi cod. Deoarece zona de date este comuna, toate variabilele procesului vor fi vazute la fel de catre toate firele de executie, orice modificare facuta de catre un fir devenind vizibila pentru toate celelalte.

Generalizand, un proces, asa cum era el perceput in lucrarile de laborator precedente, este de fapt un proces format dintr-un singur fir de executie.

La nivelul sistemului de operare, executia in paralel a firelor de executie este obtinuta in mod asemanator cu cea a proceselor, realizandu-se o comutare intre fire, conform unui algoritm de planificare.

Spre deosebire de cazul proceselor, insa, aici comutarea poate fi facuta mult mai rapid, deoarece informatiile memorate de catre sistem pentru fiecare fir de executie sunt mult mai putine decat in cazul proceselor, datorita faptului ca firele de executie au foarte putine resurse proprii.

Practic, un fir de executie poate fi vazut ca un numarator de program, o stiva si un set de registri, toate celelalte resurse (zona de date, identificatori de fisier etc) apartinand procesului in care ruleaza si fiind exploatate in comun.

In Java avem doua posibilitati de a crea un fir de executie:

* Prin definirea unei clase care mosteneste de la clasa predefinita Thread (deriva din clasa Thread)
* Prin implementarea interfetei Runnable

Clasa***Thread*** este definita in pachetul Java.lang avand o serie de metode. Este o clasa care are ca si superclasa clasa Object si implementeaza interfata Runnable.

Interfata Runnable declara o singura metoda run() si clasa Thread implementeaza aceasta. Masina virtuala Java permite unei aplicatii sa ruleze mai multe fire de executie in paralel.

Fiecare fir de executie are o prioritate si fiecare fir poate fi marcat ca si fir demon.

In momentul crearii unui fir de executie se seteaza si proprietatile acestea, adica firul nou creat va avea aceeasi prioritate ca si firul parinte si va fi de tip demon numai daca firul parinte este demon.  
            Cand se interpreteaza codul octeti al unei aplicatii, masina virtuala Java creaza un prim fir de executie care ruleaza metoda main.

Masina virtuala Java continua sa execute acest fir sau alte fire noi create pana cand toate firele de executie nedemon sunt distruse.

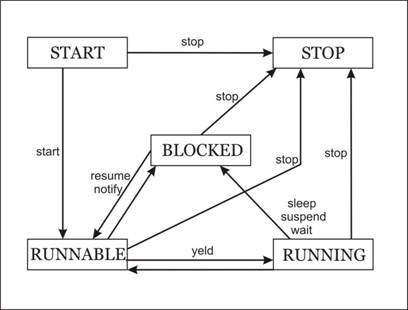
Metoda run() este “inima” oricarui fir de executie. Un fir de executie creat nu este automat pornit, lansarea sa fiind realizata cu metoda start(), definita in clasa Thread. Metoda start() se utilizeaza o singura data in ciclul de viata al unui fir.

// Creare firul de executie

**Thread fir= new Thread(“Nume fir de executie”);**

// Lansarea in executie

**fir.start();**

Firul se distruge cu metoda destroy*(),* si aceasta putand fi apelat o singura data. Metoda *sleep()*se apeleaza cu un argument care reprezinta timpul in milisecunde in care firul de executie asteapta(adoarme). *sleep()* este o metoda statica se apeleaza prefixat cu numele clasei.

Metoda statica *Thread.Sleep*permite oprirea activitatii unui fir pentru un numar de mili-secunde, numar specificat ca argument; aceasta metoda actioneaza doar asupra firului curent

(ea apeleaza intern Thread.CurrentThread).

Valoarea 0 pentru argument specifica faptul ca firul curent isi suspenda executia pentru a permite altor fire, aflate in asteptare, sa-si continue executia; valoarea *Timeout.Infinite*pentru argument specifica blocarea pe termen nedefinit a firului curent.

In caz ca firul este intrerupt se genereaza exceptia *InterruptedException*.

Deci apelul metodei trebuie facut intr-un context de tratarea acestei exceptii.

**try**

**catch ( InterruptedException e )**

            Firul poate fi scos din starea de adormire prin apelul metodei*interrupt().*

Metoda se mai apeleaza pentru a intrerupe executia unui fir care este blocat in asteptarea unei operatii lungi de intrare/iesire.

Metoda *join()* se utilizeza pentru legarea firelor de executie.

Prin *getName()*si *setName()*putem obtine sau sa setam numele firului.

            In cazul in care nu vrem sa dam nume firelor de executie, atunci renuntam la supraincarcarea constructorului si folosim constructorul implicit, fara argumente, care creeaza un fir de executie fara nume.

**Thread fir= new Thread();**

In cazul in care dorim sa cream o clasa care instantiaza fire de executie si nu putem extinde clasa Thread, atunci trebuie sa implementam direct interfata Runnable.

Clasa Thread implementeaza ea insasi aceasta interfata. Interfata Runnable asigura un protocol comun obiectelor care doresc sa-si execute codul, atata timp cat sunt active.

Un obiect este activ daca a fost lansat in executie si nu a fost oprit. Inima unui obiect, care implementeaza clasa Runnable, este metoda run(). Aceasta metoda trebuie suprascrisa de obiectul de tip Runnable.

De altfel, este singura metoda a interfetei Runnable. Interfata se numeste Runnable si nu Running, deoarece ea nu se executa chiar tot timpul.

Pe marea majoritate a masinilor se afla un singur procesor care este ocupat si cu alte sarcini. Doar o parte din timpul de lucru al procesorului este alocata obiectului de tip Runnable.

**public class Clasa implements Runnable**

**}**

Spre deosebire de modalitatea anterioara, se pierde tot suportul oferit de clasa Thread.             Simpla instantiere a unei clase care implementeaza interfata Runnable nu creeaza nici un fir de executie, crearea acestora trebuind facuta explicit.

Pentru a realiza acest lucru trebuie sa instantiem un obiect de tip Thread ce va reprezenta firul de executie propriu-zis al carui cod se gaseste in clasa noastra.

Acest lucru se realizeaza, ca pentru orice alt obiect, prin instructiunea new, urmata de un apel la un constructor al clasei Thread, insa nu la oricare dintre acestia.

Trebuie apelat constructorul care sa primeasca drept argument o instanta a clasei noastre.

Dupa creare, firul de executie poate fi lansat printr-un apel al metodei start. Aceste operatiuni pot fi facute chiar in cadrul clasei noastre:

**public class Fir implements Runnable**

**}**

**public void run()**

**}**

Specificarea argumentului this in constructorul clasei Thread determina crearea unui fir de executie care, la lansarea sa, va apela metoda run din clasa curenta.

Asadar, acest constructor accepta ca argument orice instanta a unei clase Runnable.

Pentru clasa FirExecutie, lansarea firului va fi facuta automat la instantierea unui obiect al clasei:

**Fir fir = new Fir ();**

Metoda run nu trebuie apelata explicit, acest lucru realizandu-se automat la apelul metodei start.

|  |
| --- |
|  |
|  | http://www.scritub.com/files/informatica/java/229_poze/image004.gif |

Apelul explicit al metodei run nu va furniza nici o eroare, insa aceasta va fi executata ca orice alta metoda si nu separat intr-un fir.

Fiecare fir de executie are propriul sau ciclu de viata : este creat, devine activ prin lansarea sa si la un moment dat, se termina.

Un fir de executie se poate gasi in una din urmatoarele patru stari:

* New Thread
* Runnable
* Not Runnable
* Dead

Un fir de executie se gaseste in starea New Thread imediat dupa crearea sa, adica dupa instantierea unui obiect din clasa Thread sau dintr-o subclasa a sa.

**Thread fir = new Thread(obiect);**

**// fir se gaseste in starea New Thread**

Apelul oricarei alte metode in afara de start() nu are nici un sens si va provoca o exceptie de tipul IllegalThreadStateException.

Dupa apelul metodei start() un fir va trece in starea Runnable, adica va fi in executie.

**fir.start();**

**//fir se gaseste in starea Runnable**

Metoda start() realizeza urmatoarele operatiuni necesare rularii firului de executie:

* Aloca resursele sistem necesare.
* Planifica firul de executie la procesor pentru a fi lansat.
* Apeleaza metoda run() a obiectului activ al firului.

Un fir de executie poate ajunge in stare Not Runnable in una din urmatoarele situatii:

* Este ”adormit” prin apelul metodei sleep;
* A apelat metoda wait, asteptand ca o anumita conditie sa fie satisfacuta.
* Este blocat intr-o operatie de intrare/iesire.

Pentru fiecare tip de intrare in starea Not Runnable, exista o secventa specifica de iesire din starea repectiva, care readuce firul de executie in starea Runnable.

Acestea sunt:

* Daca un fir de executie a fost ”adormit”, atunci el devine Runnable doar dupa scurgerea intervalului de timp specificat de instructiunea sleep.
* Daca un fir de executie asteapta o anumita conditie, atunci un alt obiect trebuie sa il informeze daca acea conditie este indeplinita sau nu. Acest lucru se realizeaza prin instructiunile notify sau notifyAll.
* Daca un fir de executie este blocat intr-o operatiune de intrare/iesire atunci el redevine Runnable atunci cand acea operatiune s-a terminat.

Starea Dead este starea in care ajunge un fir de executie la terminarea sa. Un fir nu poate fi oprit din program printr-o anumita metoda, ci trebuie sa se termine in mod natural la incheierea metodei run pe care o executa.

Spre deosebire de versiunile curente ale limbajului Java, in versiunile mai vechi exista metoda stop() a clasei Thread care termina fortat un fir de executie, insa aceasta a fost eliminata din motive de securitate.

Conceptul de thread (fir de executie) este folosit in programare pentru a eficientiza executia programelor, executand portiuni distincte de cod in paralel, in interiorul aceluiasi proces.

Cateodata insa, aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente si in anumite momente ale executiei, se poate intampla ca un thread sa trebuiasca sa astepte executia unor instructiuni din alt thread, pentru a putea continua executia propriilor instructiuni.

Aceasta tehnica prin care un thread asteapta executia altor threaduri inainte de a continua propria executie, se numeste s*incronizarea threadurilor*.

Java ofera urmatoarele facilitati pentru sincronizarea threadurilor:

       mecanismul *synchronized*

       si metodele: *wait, notify* si *notifyAll*.

Pentru a intelege problematica sincronizarii threadurilor, vom considera problema producatorilor si a consumatorilor.

Aceasta spune ca avem mai multi producatori care 'produc' in paralel obiecte si le depoziteaza intr-un container comun si avem mai multi consumatori care 'consuma' in acelasi timp obiectele depozitate in container de catre producatori.

Toti producatorii si consumatorii vor partaja acelasi container. Pentru a simplifica putin lucrurile am ales ca container care va fi partajat de producatori si consumatori, o clasa *Product* care incapsuleaza o valoare de tip *int*.

Astfel ca producatorii si consumatorii vor produce, respectiv consuma, valori de tip *int*. Ca sa fie si mai simplu, containerul (obiect de tipul *Product*) poate contine o singura valoare de tipul *int* (poate contine un singur produs), iar la un moment dat exista un singur consumator si un singur producator in executie.

Pornind de la aceste reguli, am scris urmatorul program pentru problema producatorilor si consumatorilor (producatorii si consumatorii sunt evident threaduri):

class Consumer extends Thread

    public void run() catch (InterruptedException e)

        }

    }

}

class Producer extends Thread

    public void run() catch (InterruptedException e)

        }

    }

}

class Product

            public boolean empty()

            public int get() catch(Exception ex)

                        }

                        available = false;

                        System.out.println('am consumat produsul '+value);

                        return value;

            }

            public void set(int val) catch(Exception ex)

                        }

                        available = true;

                        value = val;

                        System.out.println('am produs '+val);

            }

}

public class Thrsynchr

}

In program se folosesc metodele *get()* si *set()* din clasa *Product* pentru a consuma, respectiv produce, o valoare de tipul *int*.

Unul dintre bug-urile care pot aparea in cazul in care modificam programul in asa fel incat sa avem 2 consumatori care sa consume in paralel din acelasi container (care poate contine maxim o valoare de tipul int) si un singur producator care sa puna cate o valoare in acel container, este situatia in care cei doi consumatori consuma aceeasi valoare produsa o singura data de catre producator.

Acesta este un bug-urile complicat, deoarece aparitia lui este aleatoare, drept urmare, el poate fi reprodus destul de greu. Presupunem ca initial containerul e gol.

Consumatorul 1 tot executa ciclul *while (available==false)*din metoda *Product.get()* deoarece nu exista nici o valoare de consumat.

La fel si Consumatorul 2 este blocat de ciclul while pana producatorul va produce o valoare. Cat timp cei doi consumatori sunt in ciclul while, threadul Producator produce o valoare.

Sa presupunem ca primul care va iesi din while va fi Consumatorul 1.

Consumatorul 1 va consuma valoarea din container, dar sa presupunem ca inainte de a seta *available* la *false*, Consumatorul 2 iese si el din while si incepe si el sa consume acceasi valoare pe care o consuma Consumatorul 1.

In cazul exemplului nostru simplu, e drept, aceasta situatie poate avea loc mai rar (totusi ea se poate intampla!), dar in cazul programelor mai complexe care contin mai multe linii de cod intre partea de verificare a disponibilitatii datelor (ciclul wait) si partea de consum efectiv a valorii (*available=false*), acest bug poate aparea suparator de frecvent.

            In mod frecvent, actiunile intreprinse de mai multe fire de executare active se refera la unele resurse comune (variabile, obiecte, fisiere, baze de date etc.).

In acest caz pot aparea aspecte nedorite, specifice programarii concurente.

Ne rezumam la *problema excluderii reciproce*, ce reclama ca la fiecare moment de timp cel mult un fir de executare sa aiba acces la resursele comune; daca un fir incearca sa acceseze resursele comune cand un alt fir le acceseaza, el va fi blocat.

Pentru a realiza excluderea reciproca asupra operatiei de incrementare a unei variabile comune care poate fi accesata simultan de mai multe fire de executie, vom folosi facilitatile limbajului Java de a declara o metoda cu modificatorul synchronized.

            Regula este urmatoarea: daca mai multe fire de executie apeleaza metode declarate cu modificatorul synchronized prin intermediul aceluiasi obiect Ob avand ca tip clasa C, atunci la fiecare moment de timp cel mult un fir de executie este in curs de a executa o astfel de metoda.

Daca un fir de executie este in curs de a executa o metoda sincronizata, el o va executa pana la capat.

Orice alt fir de executie ce invoca o astfel de metoda va fi blocat. Cand un fir termina de executat o metoda sincronizata, atunci unul din firele de executie blocate va fi deblocat si va incepe efectiv invocarea metodei dorite.

            Este folosita urmatoarea tehnologie:  obiectul Ob se numeste monitor spunem ca monitorul este liber daca nu este in curs de executare nici o metoda sincronizata a sa.

In caz contrar, adica daca exact o metoda sincronizata a sa este in curs de executare spunem ca monitorul este ocupat.

Daca un fir de executie invoca, prin intermediul monitorului Ob o metoda sincronizata a clasei C si monitorul este ocupat, firul va fi blocat in multimea de fire blocate de asteptare asupra monitorului.

Daca un fir este in curs de a executa, prin intermediul obiectului Ob o metoda sincronizata a clasei C, spunem ca el detine controlul exclusiv asupra monitorului.

Cand firul de mai sus termina de executat o metoda sincronizata, el pierde controlul asupra monitorului.

Daca multimea asociata monitorului este nevida se alege “la intamplare” un fir din aceasta multime si acesta trece la executarea metodei pe care acesta a incercat, fara succes, sa o invoce.

Acum monitorul este din nou ocupat si noul fir este cel care detine controlul asupra monitorului.

Java permite nu numai sincronizarea unei intregi metode, dar si sincronizarea doar a unui bloc, delimitand astfel mai strict portiunea “critica” din cod, in scopul cresterii vitezei programului.

Principalele metode (primitive) de sincronizare puse la dispozitie de Java sunt urmatoarele metode publice ale clasei radacina Object:

final void wait()

final void wait(long t)

final void notify()

final void notifyAll()

Aceste metode pot lansa exceptia IllegalMonitorStateException daca firul curent nu detine controlul asupra monitorului reprezentat de obiectul curent.

Metodele trebuie sa fie invocate din interiorul unei metode de sincronizare sau a unui bloc sincronizat.

Ca urmare a executarii instructiunii wait() thread-ul curent este suspendat, sistemul de rulare Java plaseaza thread-ul intr-o coada de asteptare interna si inaccesibila programatorului.

Ca urmare a executarii instructiunii notify() din coada de asteptare interna este scos in mod arbitrar un thread.

Acest thread trebuie sa obtina blocajul de sincronizare pentru obiectul tinta, care intotdeauna va determina blocarea cel putin pana thread-ul va chema metoda notify().

Thread-ul este atunci reluat din punctul unde apare metoda wait.

Invocarea metodei notifyAll() lucreaza in acelasi mod ca si notify() numai ca pasii de mai sus se aplica la toate thread-urile ce asteapta in coada de asteptare pentru obiectul tinta. Doua versiuni alternative ale metodei wait() preia, argumente specificand timpul maxim de asteptare in coada.

Daca timpul de asteptare este depasit, atunci metoda notify() este invocata automat. Daca o instructiune interrupt() apare in timpul executiei unei instructiuni wait(), acelasi mecanism notify() se aplica exceptand controlul intors catre clauza catch asociata cu invocarea lui wait().

            Gruparea firelor de executie pune la dispozitie un mecanism pentru manipularea acestora ca un tot si nu individual.

            De exemplu, putem sa pornim sau sa suspendam toate firele dintr-un grup cu un singur apel de metoda. Gruparea firelor de executie se realizeaza prin intermediul clasei **ThreadGroup**.

            Fiecare fir de executie Java este membru al unui grup, indiferent daca specificam explicit acest lucru.

Afilierea unui fir de executie la un anumit grup se realizeaza la crearea sa si devine permanenta, in sensul ca nu vom putea muta un fir de executie dintr-un grup in altul, dupa ce acesta a fost creat.

In cazul in care cream un fir de executie fara a specifica in constructor din ce grup face parte, el va fi plasat automat in acelasi grup cu firul de executie care l-a creat.

La pornirea unui program Java se creeaza automat un obiect de tip ThreadGroup cu numele **main**, care va reprezenta grupul tuturor  firelor de executie create direct din program si care nu au fost atasate explicit altui grup.

Cu alte cuvinte, putem sa ignoram complet plasarea firelor de executie in grupuri si sa lasam sistemul sa se ocupe cu aceasta, adunandu-le pe toate in grupul main.

Exista situatii cand programul creeaza multe fire de executie, iar gruparea lor poate usura substantial manevrarea lor. Un grup se compune dintr-o multime de thread-uri si poate contine alte grupuri, formand un arbore, un thread poate accesa doar informatia referitoare la propriul grup fara a avea acces la alte grupuri sau la grupul parinte.

Orice thread apartine unui grup specificat ca un parametru constructor sau mostenit.            Thread-urile unui grup pot fi enumerate, pentru un grup poate fi setata prioritatea maxima respectiv grupul de thread-uri poate fi intrerupt din executie.

           Grupurile de thread-uri sunt folosite deoarece permit manipularea mai multor thread-uri la un singur apel de functie si ofera bazele mecanismului de securitate folosit pentru lucrul cu thread-uri. In mod implicit toate thread-urile create in program apartin unui grup definit de masina virtuala Java.

Clasa ce manipuleaza grupurile de threaduri este *ThreadGroup*, ale carei metode sunt:

**public class ThreadGroup**

            Unele dintre metodele de manipulare a grupurilor de thread-uri sunt preluate de la clasa Thread, prin extinderea semnificatiei pentru grupuri.

Un constructor de thread, in forma cea mai generala ataseaza noul thread unui grup de thread-uri, daca argumentul lipseste atunci este atasat grupului curent.

Grupul unui thread este stabilit la crearea acestuia si nu mai poate fi modificat pe parcursul executiei sale, la terminare thread-ul este automat eliminat din grup.

Pentru a afla carui grup apartine un anumit fir de executie putem folosi metoda getThreadGroup a clasei Thread.

Un grup poate avea ca parinte un alt grup, ceea ce inseamna ca firele de executie pot fi plasate intr-o ierarhie de grupuri, in care radacina este grupul implicit main.

3         Versiunea JDK 5.0 a fost un pas major in programarea concurenta astfel masina virtuala java a fost imbunatatita semnificativ pentru a permite claselor sa profite de suportul pentru concurenta oferit la nivel hardware.

           De asemenea, un set bogat de clase noi au fost adaugate pentru a face usoara dezvoltarea de aplicatii concurente.

Pachetul *java.util.concurent* aduce un set bine testat si foarte performant de blocuri pentru concurenta.

Crearea acestui framework pentru concurenta echivaleaza cu crearea framework-ului *Collections* pentru structurile de date.

Pachetul *java.util.concurrent* adauga cateva noi colectii concurente: *ConcurrentHashMap, CopyOnWriteArrayList* si *CopyOnWriteArraySet*.

Scopul acestor clase este sa imbunatateasca performanta si scalabilitatea oferita de tipurile de baza de colectii.

JDK 5.0 ofera de asemenea doua noi interfete de colectii: *Queue*si *BlockingQueue*.

Iteratorii s-au schimbat de asemenea in JDK 5.0.

Daca pana la versiunea 5.0 nu se permitea modificarea unei colectii in timpul iteratiei, iteratorii introdusi in JDK 5.0 ofera un view consistent asupra colectiei, chiar daca aceasta se schimba in timpul iterarii.

*CopyOnWriteArrayList* si *CopyOnWriteArraySet*sunt versiuni imbunatatite ale Vector si ArrayList. Imbunatatirile sunt aduse in special la nivelul iteratiei.

Astfel, daca in timpul parcurgerii unui Vector sau a unui ArrayList colectia este modificata, se va arunca o exceptie. Noile clase rezolva aceasta problema.

*Queue*

Exista doua implementari principale, care determina ordinea in care elementele sunt accesate: ConcurrentLinkedQueue (acces FIFO) si PriorityQueue (acces pe baza de prioritati)

*BlockingQueue*

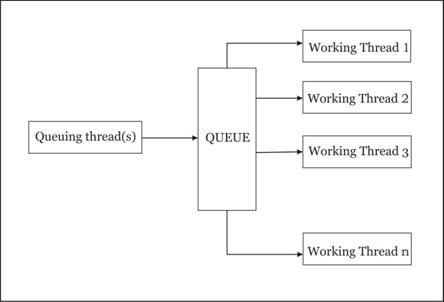
Sunt folosite atunci cand se doreste blocarea unui thread cand anumite operatii pe o coada nu pot fi executate. Un exemplu ar fi cazul in care consumatorii scot mai greu din coada informatia decat ea este pusa in coada de producatori.

Prin folosirea *BlockingQueue* se blocheaza automat producatorii pana cand se elibereaza un element din coada.

Implementari ale interfetei BlockingQueue sunt:

**LinkedBlockingQueue**, **PriorityBlockingQueue, ArrayBlockingQueue**

**si SynchronousQueue**.



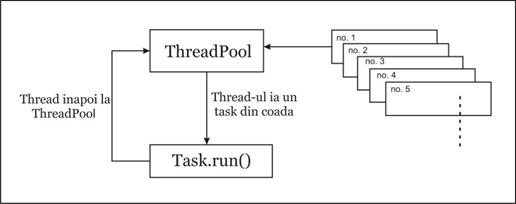
**Refolosirea thread-urilor: Thread Pools**

Un mecanism clasic pentru managementul unui grup mare de task-uri este combinarea unui *work queue* cu un *thread pool*.

Un *work queue* este o coada de taskuri ce trebuie procesate. Un *thread pool* este o colectie de thread-uri care extrag sarcini din coada si le executa.

Cand un *worker thread* termina o sarcina, se intoarce la coada pentru a vedea daca mai exista sarcini de executat.

Daca da, scoate sarcina din coada si o executa.



***Framework-ul Executor***

Pachetul *java.util.concurrent*contine un intreg framework pentru manangementul executiei task-urilor care implementeaza *Runnable.*

Un aspect foarte important al folosirii framework-ului *Executor* este faptul ca se permite decuplarea submiterii task-urilor de politica de executie.

Aceasta permite schimbarea politicii de executie foarte usor, fara a fi nevoie de modificari majore in cod.

**Interfata *Executor*** este foarte simpla:

public interface Executor

Politica de executie a task-urilor depinde de implementarea de *Executor* aleasa.

Clasa *Executors* ofera diverse metode statice pentru obtinerea de instante de diferite implementari de executori.

**Executors.newCachedThreadPool()**

**Executors.newFixedThreadPool(int n)**

**Executors.newSingleThreadExecutor()**

Executorii returnati de primele doua metode sunt instante ale clasei ThreadPoolExecutor, care poate fi intens customizata in functie de necesitati.

4.

            Programarea retelelor este de obicei descrisa folosind conceptele de client si server. Un server este o aplicatie care ruleaza pe un calculator gazda (host) care furnizeaza o conexiune si informatii utile din momentul stabilirii acelei conexiuni.

Un client este o aplicatie care ruleaza pe un calculator din retea, cautand sa stabilieasca o conexiune cu un server.

De regula se pot conecta mai multi clienti simultan la un server.

            Serverul este un furnizor de informatii, iar clientul un consumator. Clientul are nevoie de informatii, se conecteaza la un server caruia ii adreseaza o cerere.

Respectivul server trimite inapoi un raspuns care poate sa contina informatia ceruta sau ii indica faptul ca nu poseda respectiva informatie.

Clientul poate sa continue emiterea de cereri , eventual altor servere.

            Pentru a se putea realiza o comunicare intre clienti si server, acestia vor trebui sa respecte un set de reguli (protocol).

De exemplu mai intai trimite clientul cererea, dupa care i se trimite un raspuns. Mesajele trimise trebuie sa fie intelese de cei doi parteneri.

Calculatoarele conectate in retea comunica intre ele utilizand doua protocoale:

       Orientat spre conexiune  TCP (Transmission Control Protocol)

       Neorientat spre conexiune  UDP (User Datagram Protocol)

Internetul foloseste *nume (adrese) simbolice* pentru retele si pentru masinile gazda; ele se mai numesc *nume de domeniu*.

Lor li se asociaza in mod biunivoc *adrese (numerice) IP*, folosite efectiv la comunicarea intre nodurile retelei.

Asocierea cade in sarcina unui *sistem de nume de domeniu* (*DNS* = Domain Name System).

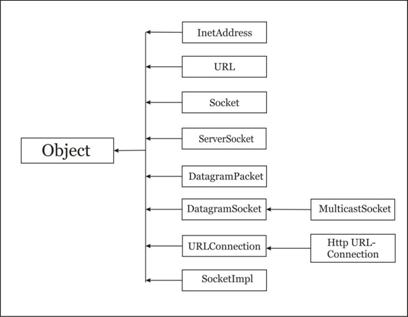
Protocolul TCP este un protocol orientat spre conexiune aflat pe nivelul transport care asigura servicii de comunicare sigure cu detectarea si corectarea erorilor intre doua gazde.  Serverele sunt cele care isi incep primele activitatea.

In mod tipic, un server ofera succesiv clientilor posibilitatea de a se conecta la el (spunem ca *accepta conexiuni* de la clienti).

La inceput clientul isi manifesta dorinta de a se conecta si daca serverul este gata sa accepte conexiunea, aceasta se realizeaza efectiv; este vorba deci de o actiune de la client catre server.

Apoi transmisia informatiilor devine bidirectionala, fluxul de informatii putand circula acum in ambele sensuri.

Teoretic, activitatea unui server se desfasoara la infinit.

Pentru conectare la server, clientul trebuie sa cunoasca adresa serverului (fie cea numerica, fie cea simbolica), precum si numarul *portului* pus la dispozitie de server.

Portul nu este o locatie fizica, ci o extensie software corespunzatoare unui serviciu. Serverul poate oferi mai multe servicii, pentru fiecare fiind alocat un (numar de) port. Combinatia dintre adresa IP si numarul de port este folosita pentru crearea unui termen abstract, numit **socket** (canal de comunicatie).

Un socket furnizeaza facilitati pentru crearea de fluxuri de intrare/iesire, care permit schimburile de date intre client si server.

Atunci cand se stabileste o conexiune, atat clientul cat si serverul vor avea cate un socket, comunicarea efectiva realizandu-se intre socketuri.

Principalele operatii care sunt realizate de socket-uri sunt:

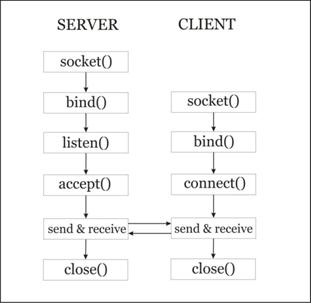
       conectare la un alt socket

       trimitere date

       receptionare date

       inchidere conexiune

       acceptare conexiuni

Clasa ServerSocket din pachetul java.net implementeaza operatiile legate de starea de server a socketului. Fara ea nu s-ar putea scrie aplicatii de tip client/server in Java.

Constructorii acestei clase sunt:

       public ServerSocket(int port) throws IOException - Creeaza un serversocket pe portul specificat. In cazul aparitiei unei erori la deschiderea socketului este captata exceptia IOException.

       public ServerSocket(int port, int count)throws IOException - Creaza un serversocket pe un port dat cu un timp maxim de ascultare. Se poate conecta pe portul 0 ca anonymous.

Dintre metodele acestei clase amintim pe cele mai importante:

       public InetAddtress getInetAddress() - Intoarce adresa la care socketul este conectat.

       public int getLocalPort() - Intoarce portul pe care asculta socketul.

       public Socket accept() throws IOException - Accepta o conexiune. Aceasta metoda va bloca sistemul pana cand conexiunea va fi realizata efectiv.

       public void close() throws IOException - Sistemul inchide server socketul. Daca apare o eroare in timpul inchiderii socketului, atunci este captata exceptia IOException.

Structura generala a serverului este urmatoarea:

1.   Creeaza un obiect de tip ServerSocket la un anumit port

while (true)

            Un mod tipic de creare a unui *socket server* in Java este urmatorul:

**ServerSocket ss = null; Socket cs = null;**

**try**

**catch(IOException e)**

**InputStream is = null; OutputStream os = null;**

**try**

**catch(UnknownHostException e)**

**catch(IOException e)**

Observam ca pentru server nu este necesara precizarea unei adrese IP, ci numai a unui port. Acesta trebuie sa coincida cu cel folosit de client.

Metoda accept lucreaza astfel: se asteapta ca un client sa incerce sa se lege la server pe portul precizat; in momentul in care acest lucru se intampla si legatura s-a stabilit, metoda creeaza si intoarce un socket cs de tip client.

Acestuia ii atasam un flux de intrare si unul de iesire.

Ca si pentru client, fluxurile pot fi inzestrate cu facilitatile de transmitere a datelor primitive, a obiectelor etc.

            In final toate socket-urile si fluxurile trebuie inchise explicit prin invocarea metodei close.

Structura generala a aplicatiei client este urmatoarea:

1. Citeste sau declara adresa IP a serverului si portul la care acesta ruleaza;

2. Creeaza un obiect de tip Socket cu adresa si portul specificate;

3. Comunica cu serverul:

3.1 Deschide un flux de iesire si trimite cererea;

3.2 Deschide un flux de intrare si primeste raspunsul;

3.3 Inchide fluxurile si socketul creat;

Un mod tipic de creare a unui *socket client* in Java este urmatorul:

**Socket cs = null;**

**InputStream is = null; OutputStream os = null;**

**try**

**catch(UnknownHostException e)**

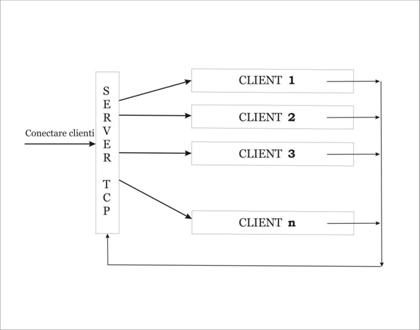
**catch(IOException e)**

unde adresa este adresa IP a serverului, iar nrPort este numarul portului ales pentru comunicare.

Socket-ului ii sunt atasate fluxul os ce va fi folosit pentru a transmite date serverului, precum si fluxul is ce va fi folosit pentru receptionarea datelor transmise de server.

Dupa caz, fluxurile vor fi inzestrate cu facilitatile de transmitere a datelor primitive, a obiectelor etc.

Un server este capabil sa comunice cu mai multi clienti. Actiunea clientilor se rezuma la conectarea lor la server:

                                                                                  import java.net.\*;

 import java.util.\*;

class ManyClients

}

In esenta serverul porneste cate un fir de executare pentru fiecare client care se conecteaza la el.

Mai precis serverul creaza un socket de server ss, care in mod repetat asteapta ca un client sa se conecteze cu succes; cand acest lucru se realizeaza, este obtinut un socket de client cs si este pornit un fir de executare caruia ii este comunicata referinta la cs si identitatea clientului (al catelea este el conectat la server); apoi serverul reia asteptarea conexiunii de la alt client.

Firul nou creat este anonim.

Ca urmare actiunea care trebuie sa o intreprinda cade in sarcina constructorului. Firele corespunzatoare clientilor foloseasc o  resursa comuna, ceea ce conduce la probleme de concurenta.

4.     **CATEVA CUVINTE DESPRE INVOCAREA METODELOR LA DISTANTA**

Java RMI (Remote Method Invocation) furnizeaza posibilitatea obiectelor Java, care se afla pe masini diferite, sa comunice utilizand apeluri obisnuite de metode.

Aceasta permite programatorului sa evite utilizarea directa a protocoalelor de comunicare intre aplicatii (TCP,UDP,etc) si sa foloseasca in schimb metode de nivel inalt. Facilitatile Java pentru invocarea la distanta sunt grupate in pachetul java.rmi.

            Spre deosebire de programarea complexa, bazata pe socketuri, mecanismul RMI ofera un context de programare distribuita flexibil si relativ simplu, reprezentand mecanismul suport pentru diverse alte modele de dezvoltare a aplicatiilor distribuite Java.

Un sistem de programare a aplicatiilor distribuite trebuie sa ofere dezvoltatorului de aplicatii urmatoarele facilitati:

       localizarea obiectelor la distanta - aplicatiile folosesc diverse mecanisme pentru a obtine referintele obiectelor remote (serviciul de nume rmiregistry sau transferul referintelor la obiecte ca parte a operatiei solicitate)

       comunicare cu obiectele remote - detaliile comunicarii sunt gestionate de RMI, pentru programator comunicatia remote este similara unei invocari de metode locale

       incarcarea codului intermediar al clasei pentru obiecte transferate ca parametri sau valori intoarse. Deoarece permite ca apelantul sa transfere obiecte, RMI integreaza mecanismele necesare incarcarii codului obiectului si transmiterea datelor.

            Serverul apeleaza un sistem de localizare a obiectelor (registry) pentru a asocia un nume cu un obiect remote, astfel clientul va cauta obiectul dupa nume in registry si apoi va invoca metoda.

Pot fi utilizate servere Web pentru a incarca codul intermediar al claselor Java client-server pentru obiectele aplicatiei daca este necesar folosind orice protocol de tip URL suportat de platforma Java.

           Arhitectura sistemului RMI integreaza 3 niveluri: stratul stub/skeleton, stratul de referinte la distanta si stratul de transport, fiecare nivel software fiind delimitat de o interfata si un protocol specifice, fiind independent de celelalte si posibil inlocuibil cu o implementare alternativa.

Pentru interactiunile la distanta sunt folosite doua tehnici astfel: pentru a realiza transmiterea transparenta a unui obiect intre spatii de adrese diferite, e folosita tehnica de serializare a obiectului, iar pentru a permite clientului apelul metodelor la distanta e utilizata tehnica de incarcare dinamica a unui intermedia de acces (stub) ce implementeaza acelasi set de interfete la distanta ca si obiectul.

*Stub*-ul este o clasa care translateaza automat apelurile metodelor la distanta in comunicatii de retea si trimitere de parametri. *Skeleton*-ul este clasa care se afla pe server si accepta conexiuni de retea, pe care le translateaza in apeluri de metode actuale pe obiectul actual.

Serverul RMI trebuie sa implementeze interfata remote, deci si metodele acestei interfete.

El poate implementa si alte metode, dar numai metodele definte in interfata pot fi apelate la distanta.

Obiectele la distanta mostenesc clasa java.rmi.server.UnicastRemoteObject.

Pe langa implementarea metodelor interfetei remote, serverul trebuie sa inregistreze (*binding*) obiectele la distanta in registrul de nume.

Un *registru de nume*pune la dispozitie facilitati de stocare si regasire a referintelor la obiectele aflate la distanta.

Un astfel de registru este initializat la executarea comenzii rmiregistry. Registrul de nume furnizeaza un serviciu simplu de nume, care pune in corespondenta o referinta a obiectului la distanta cu numele obiectului.

Drept urmare, clientul poate obtine un obiect la distanta cunoscand doar numele sau. Registrul este pornit pe masina server, dar si clientul trebuie sa aiba acces la el. Ca orice instrument bazat pe TCP/IP, registrul asteapta cererile la un port. Portul implicit este 1099.

Accesarea registrului pentru inscrierea sau regasirea unui obiect se face prin intermediul metodelor clasei Naming din pachetul java.rmi.

Clientul trebuie sa configureze un manager de securitate ( *security manage*r ) deoarece stub-ul trebuie descarcat de la server si el va accesa registrul de nume pentru a obtine referintele obiectelor la distanta.

            Pentru transmiterea efectiva la distanta a parametrilor si a rezultatelor, RMI foloseste mecanismul de serializare (Object Serialization Protocol), protocol ce se bazeaza pe mecanismul obisnuit de serializare, la care se adauga un mecanism de adnotare a claselor.

Mecanismul prin care un obiect este transformat intr-un flux de octeti, si mai apoi reconstituit, se numeste serializare, fiind deosebit de util pentru RMI si alte aplicatii in retea.         In Java, orice obiect care implementeaza interfata java.io.

Serializaile poate fi transformat intr-un flux de octeti si mai apoi reconstituit din acesta.

Aceasta interfata nu adauga nici o metoda necesar a fi implementata, ea este o interfata atasabila ce poate fi utilizata pentru a permite serializarea claselor, in acest mod pot fi transferate obiecte complexe Java in retea, fie ca argumente ale unei metode, fie ca valori returnate.

       Orice tip pe care il folosim ca parametru sau valoare returnata la apelul unei metode la distanta trebuie sa fie serializabil.

Clasa XMLEncoder este o alternativa complementara a stream-ului ObjectOutputStream si poate si folosita pentru a genera o reprezentare textuala a JavaBean asa cum ObjectOutputStream poate fi folosita pentru a crea o reprezentare binara a obiectelor Serializable.

Despite the similarity of their APIs, the XMLEncoder class is exclusively designed for the purpose of archiving graphs of *JavaBean*s as textual representations of their public properties. Like Java source files, documents written this way have a natural immunity to changes in the implementations of the classes involved. The ObjectOutputStream continues to be recommended for interprocess communication and general purpose serialization.

The XMLEncoder class provides a default denotation for *JavaBean*s in which they are represented as XML documents complying with version 1.0 of the XML specification and the UTF-8 character encoding of the Unicode/ISO 10646 character set. The XML documents produced by the XMLEncoder class are:

* *Portable and version resilient*: they have no dependencies on the private implementation of any class and so, like Java source files, they may be exchanged between environments which may have different versions of some of the classes and between VMs from different vendors.
* *Structurally compact*: The XMLEncoder class uses a *redundancy elimination* algorithm internally so that the default values of a Bean's properties are not written to the stream.
* *Fault tolerant*: Non-structural errors in the file, caused either by damage to the file or by API changes made to classes in an archive remain localized so that a reader can report the error and continue to load the parts of the document which were not affected by the error.

Below is an example of an XML archive containing some user interface components from the *swing* toolkit:

<?xml version='1.0' encoding='UTF-8'?>

<java version='1.0' class='java.beans.XMLDecoder'>

<object class='javax.swing.JFrame'>

<void property='name'>

<string>frame1</string>

</void>

<void property='bounds'>

<object class='java.awt.Rectangle'>

<int>0</int>

<int>0</int>

<int>200</int>

<int>200</int>

</object>

</void>

<void property='contentPane'>

<void method='add'>

<object class='javax.swing.JButton'>

<void property='label'>

<string>Hello</string>

</void>

</object>

</void>

</void>

<void property='visible'>

<boolean>true</boolean>

</void>

</object>

</java>

The XML syntax uses the following conventions:

* Each element represents a method call.
* The 'object' tag denotes an *expression* whose value is to be used as the argument to the enclosing element.
* The 'void' tag denotes a *statement* which will be executed, but whose result will not be used as an argument to the enclosing method.
* Elements which contain elements use those elements as arguments, unless they have the tag: 'void'.
* The name of the method is denoted by the 'method' attribute.
* XML's standard 'id' and 'idref' attributes are used to make references to previous expressions - so as to deal with circularities in the object graph.
* The 'class' attribute is used to specify the target of a static method or constructor explicitly; its value being the fully qualified name of the class.
* Elements with the 'void' tag are executed using the outer context as the target if no target is defined by a 'class' attribute.
* Java's String class is treated specially and is written <string>Hello, world</string> where the characters of the string are converted to bytes using the UTF-8 character encoding.

Although all object graphs may be written using just these three tags, the following definitions are included so that common data structures can be expressed more concisely:

* The default method name is 'new'.
* A reference to a java class is written in the form <class>javax.swing.JButton</class>.
* Instances of the wrapper classes for Java's primitive types are written using the name of the primitive type as the tag. For example, an instance of the Integer class could be written: <int>123</int>. Note that the XMLEncoder class uses Java's reflection package in which the conversion between Java's primitive types and their associated 'wrapper classes' is handled internally. The API for the XMLEncoder class itself deals only with Objects.
* In an element representing a nullary method whose name starts with 'get', the 'method' attribute is replaced with a 'property' attribute whose value is given by removing the 'get' prefix and decapitalizing the result.
* In an element representing a monadic method whose name starts with 'set', the 'method' attribute is replaced with a 'property' attribute whose value is given by removing the 'set' prefix and decapitalizing the result.
* In an element representing a method named 'get' taking one integer argument, the 'method' attribute is replaced with an 'index' attribute whose value the value of the first argument.
* In an element representing a method named 'set' taking two arguments, the first of which is an integer, the 'method' attribute is replaced with an 'index' attribute whose value the value of the first argument.
* A reference to an array is written using the 'array' tag. The 'class' and 'length' attributes specify the sub-type of the array and its length respectively.

The XMLDecoder class is used to read XML documents created using the XMLEncoder and is used just like the ObjectInputStream. For example, one can use the following fragment to read the first object defined in an XML document written by the XMLEncoder class:

XMLDecoder d = new XMLDecoder(

new BufferedInputStream(

new FileInputStream('Test.xml')));

Object result = d.readObject();

d.close();

**PREZENTAREA APLICATIEI**

            Messengerul Instant este o aplicatie realizata in limbajul Java care permite socializarea cu prietenii care sunt online, expediandu-le mesaje instantanee si trimitandu-le fisiere.

        Features:

       Aplicatia permite logarea utilizatorilor folosind un nume si o parola. La prima logare detaliile despre cont sunt serializate si salvate in fisier, iar la fiecare autentificare se verifica daca parola introdusa este aceeasi cu cea salvata la prima logare.

       Nu este permisa logarea unui client de pe doua masini diferite in aceeasi sesiune. In cazul in care user-ul este deja online si doreste autentificarea cu acelasi cont de pe alta masina va fi anuntat ca nu este permisa decat o singura instantiere.

       In cazul in care un client foloseste pentru prima data aplicatia si nu are un nume si o parola, i se va permite crearea unui cont.

       Dupa autentificare fiecare client va primi o lista cu userii logati in acel moment cu care poate conversa. Conversatiile se realizeaza user - to - user, nu ca o conferinta. Doi utilizatori converseaza independent de alte discutii pe care le au fiecare cu alti utilizatori logati.

       La fiecare autentificare un user va primi un avatar default pe care ulterior il poate schimba cu orice tip de poza (jpeg, png, gif, bmp). Daca un utilizator isi va schima avatarul, acesta va aparea schimbat in toate ferestrele celorlalti utilizatori. Daca poza este de tip gif si contine o animatie, aceasta va fi animata si in ferestrele celorlalti useri logati.

       Clientii isi pot trimite fisiere, aceasta realizandu-se secvential. Un user alege un fisier pe care vrea sa il trimita unui alt user. Acesta din urma va fi anuntat ca primeste un fisier de la un user. Fiecare dintre cei doi va avea un prograss bar care va arata cat din dimensiunea fisierului s-a download-at. La terminarea download-arii cel care a trimis va primi un mesaj cum ca fisierul s-a trimis cu succes, iar celalalt va avea doua optiuni: fie deschide fisierul (aceasta se va deschide cu programul default al sistemului de operare instalat pe calculatorul clientului care primeste fisierul), fie inchide fereastra ce contine numele fisierul, sender-ul si progress bar-ul.

       In cadrul unei sesiuni, daca 2 clienti converseaza, replicile lor raman salvate si pot fi vizualizate pana cand acestia se deconecteaza.