Tehnologii și Aplicații

în

Calculul Paralel și Distribuit

Cosmina Ivan

Editura UTPRESS
Cluj-Napoca, 2019
ISBN 978-606-737-390-5

Editura U.T.PRESS
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999
e-mail: utpress@biblio.utcluj.ro
http://biblioteca.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia: Prof.Dr.Ing. Vasile-Teodor Dădârlat

ISBN 978-606-737-390-5

Lucrarea este destinată ca suport teoretic necesar activităților practice din cadrul disciplinei Calcul Paralel și Distribuit, disciplină ce face parte din pregătirea tehnică de specialitate a studenților Facultății de Automatică și Calculatoare, secția de Calculatoare, însă poate constitui un ghid util celor interesați de soluții de rezolvare paralelă a diverșilor algoritmi, respectiv în aplicarea unor tehnologii din domeniul calculului distribuit pentru conceperea unor aplicații distribuite.

Sunt propuse un număr de 11 lucrări, structurate pe cele două direcții de interes ale disciplinei și anume calculul paralel, respectiv calculul distribuit. Astfel, primele cinci lucrări prezintă concepte și tehnologii pentru dezvoltarea aplicațiilor paralele, respectiv următoarele cinci lucrări abordează tehnologii și algoritmi din domeniul calculului distribuit, conținutul urmărind curicula disciplinei de Calcul Paralel și Distribuit.

O primă lucrare tratează suportul pentru programarea cu threaduri multiple oferit de Java, ca abordare de bază în construcția de cod paralel, lucrarea a doua prezintă Frameworkul de concurență Java ce oferă mecanisme și structuri de date evoluate pentru concurență și paralelism. Cea de a treia lucrare prezintă modelul și biblioteca OpenMP utilizată pentru programarea mașinilor cu memorie partajată, iar lucrarea a patra prezintă MPI - model și standard de programare a aplicațiilor paralele pentru mașini cu memorie distribuită, cu largă utilizare în domeniul calculului științific de înaltă performanță (HPC) și simulărilor masive. În cea de a cincea lucrare este prezentată tehnologia CUDA pentru arhitecturi ce dispun de plăci grafice NVIDIA și susțin paralelismul masiv de date. În partea a doua, se prezintă tehnologiile de bază utilizate pentru dezvoltarea de aplicații distribuite și anume socketuri, obiecte distribuite și acces RMI, alături de tehnologii de mesagerie distribuită (JMS/RabitMQ), utilizate în implementarea de algoritmi distribuiți (în secțiunea 8 au fost aleși algoritmi distribuiți reprezentativi cum ar fi ceasuri logice, identificare leader și algoritmi pentru excludere mutuală distribuită). O ultimă lucrare prezintă o tehnologie recentă Apache-Spark ce permite procesare masivă de date (tip big data) în manieră de execuție paralelă în medii distribuite.

Fiecare lucrare este structurată în patru secțiuni principale. Astfel, o primă secțiune prezintă obiectivele de studiu, cea de a doua secțiune oferă cunoștințele teoretice ce vizează aspectele tehnologice necesare pentru a rezolva cerințele practice, iar ultimele două secțiuni prezintă aspecte practice de implementare, propun întrebări recapitulative și exerciții, respectiv taskuri sau miniproiecte mai complexe, având drept scop aprofundarea și aplicarea directă a cunoștințelor teoretice prezentate.

Pentru activitățile practice se vor putea consulta complementar fiecărei lucrări, exemplele implementate ce reprezintă soluții oferite problemelor propuse și care pot fi descărcate de la adresa ***https:/ftp.utcluj.ro /~civan/CPD***, exemple ce au fost dezvoltate în decursul ultimilor ani cu ajutorul studenților seriei în limba românî din anul IV ai secției de Calculatoare, iar pe această cale doresc să le aduc mulțumiri sprijinului oferit.

Cluj- Napoca, 30 iunie 2019

Cuprins

1.Programare multithread

Programarea multithread constituie o caracteristică de bază a limbajului Java, ce permite crearea de obiecte multiple care pot executa simultan diferite operații, oferind astfel suport pentru concurență și paralelism în aplicații.

1.1.Obiective

- Studiul mecanismelor de programare cu threaduri multiple în Java
- Implementarea unor aplicații ce presupun concurență/paralelism prin crearea și utilizarea mai multor threaduri

1.2. Concepte

Firele de execuţie (engl. threaduri) fac trecerea de la programarea secvenţială la programarea concurentă/paralelă. Java integrează mecanismele programării concurente sub forma unor clase şi interfeţe specifice, oferind astfel suportul necesar programării aplicaţiilor moderne ce necesită threaduri multiple. Se oferă astfel suport pentru creşterea performanţelor aplicaţiilor prin implementarea *paralelismului real* pentru sisteme *multiprocesor*, respectiv a *paralelismului prin întreţeserea threadurilor* pentru sisteme monoprocesor.

Suportul multithread este implementat de maşina virtuală Java (JVM) , iar manifestarea threadurilor este direct influenţată de modul în care sistemul de operare gestionează threadurile multiple,diferit pentru diverse platforme.Un sistem de operare monotasking nu este capabil să execute decât un singur proces la un moment dat, în timp ce un sistem de operare multitasking poate rula oricâte procese în acelaşi timp concurent, alocând periodic *cuante* din timpul de lucru al CPU fiecărui proces. Am reamintit acest lucru deoarece noţiunea de fir de execuţie nu are sens decât în cadrul unui sistem de operare multitasking. Un fir de execuţie este similar unui proces secvenţial în sensul că are un început, o secvenţa de execuţie şi un sfârşit. Un program îşi poate defini însă nu doar un fir de execuţie ci oricâte, ceea ce înseamnă că în cadrul unui proces se pot executa simultan mai multe fire de execuţie, permiţând astfel execuţia concurentă a sarcinilor independente ale programului.

Execuţia simultană a firelor de execuţie în cadrul unui proces este similară cu execuţia concurentă a proceselor: sistemul de operare va aloca ciclic cuante din timpul procesorului fiecărui fir de execuţie până la terminarea lor. Deosebirea majoră între proces și thread constă în faptul că acestea din urmă nu pot exista decât în cadrul unui proces, astfel la crearea unui nou proces fiu este realizată o copie a procesului părinte: cod + date, iar la crearea unui fir de execuţie nu este copiat decât codul procesului părinte; toate firele de execuţie au deci *acces la aceleaşi date,* datele procesului original. Deasemenea, comutarea de context și comunicarea între threaduri este mult mai simplă și mai rapidă.

Proiectarea şi implementarea aplicaţiilor folosind threaduri oferă o serie de avantaje, astfel:

- simplificarea programării unor aplicații mai complexe şi o mai bună structurare a codului
- eficienţă în utilizarea resurselor sistemului (modelul multithread poate înlocui soluţiile clasice de comunicare între procese de tip IPC pentru anumite probleme)
- ascunderea latenţei acceselor la memorie, I-O sau comunicaţiei şi astfel câştig de performanţă prin suprapunere comunicaţie-procesare
- îmbunătăţirea interacţiunii la nivel de aplicaţie prin proiectarea unor interfeţe performante, responsive controlate printr-un thread GUI distinct
- sistemele moderne de obiecte distribuite sunt sisteme multithread (orice server de obiecte distribuite este multithread)
- suport pentru planificare şi echilibrarea încărcării de execuţie prin maparea dinamică a taskurilor (asociate unor threaduri) la procesoare (pentru sistemele actuale cu procesoare multiple)

1.2.1.Clasele programării Java multithread

Aplicațiile sunt structurate în procese şi astfel un proces master creează alte procese explicit pentru a putea separa logic funcționalitățile aplicației. Dacă procesele pot fi considerate construcții ce deteremină arhitectura aplicației, threadurile sunt construcții specifice limbajului.

Clasa Obiect . Următoarele metode ale clasei Object şi anume *wait(), notify() şi notifyAll(),* sunt utilizate în programarea multithread, având următoarea semnificaţie:

- *wait()* - pune obiectul în aşteptare până la apariţia unui eveniment (notificare) cu/fără indicarea duratei maxime de aşteptare
- *notify()* - permite anunţarea altor obiecte de apariţia unui eveniment
- *notifyAll()* - implementează mod broadcast notificarea mai multor obiecte la apariţia unor evenimente
- *finalize()* - poate fi suprascrisă în clase şi defineşte diversele acţiuni ce vor fi executate de "colectorul de deşeuri" JVM înaintea distrugerii obiectului

Clasa Thread. Maşina virtuală Java permite definirea mai multor threaduri concurente, orice thread este un obiect Java ce posedă metode şi poate fi transferat ca parametru, plasat într-un tablou, etc. Maşina virtuală Java realizează maparea obiectului Java de tip runnable unei implementări de thread dependentă de sistem, iar sistemul de operare îi alocă resurse. Orice obiect de tip thread posedă o metodă *run()* prin intermediul căreia se implementează manifestarea threadului, definind o buclă ce se execută până la terminarea acestuia.

Prin intermediul atributelor unui thread se pot defini caracteristici specifice acestuia şi anume: *identificatorul de thread, numele, politica de planificare, prioritatea*, cine este responsabil cu controlul execuţiei threadului (procesul utilizator sau nucleul sistemului de operare). Un thread există în cadrul unui proces, el este compus din context, structura utilizator conţinând copii ale valorilor regiştrilor generali, stiva, *zona de date private* şi *setul de instrucţiuni*. Un thread este o entitate planificabilă, ea poate fi întreruptă preemptiv şi poate utiliza în funcţie de numărul de procesoare din sistem concurenţa reală (multiprocesor) sau logică (monoprocesor). Cele mai importante proprietăți ale unui thread sunt:

- threadurile posedă nume pentru identificare, însă pot exista mai multe threaduri cu acelaşi nume, dacă numele nu a fost specificat la creare, se va genera un nume nou, implicit, acesta putând fi citit cu metoda getName
- orice nou thread începe execuţia cu *metoda run()* similar începerii execuţiei unui program cu metoda main()
- orice thread posedă **o** *anumită prioritate* (valoare întreagă între Thread.MIN_PRIORITY şi Thread.MAX_PRIORITY,în rangul 1-10,în mod curent este utilizat Thread.NORM_PRIORITY,având valoarea 5), prioritatea threadului nou creat este identică cu a threadului părinte. Maşina virtuală mapează priorităţile threadurilor nivelurilor de prioritate ale platformei care poate dispune de mai puţine niveluri de prioritate (sub Windows, anumite niveluri JVM vor fi mapate aceloraşi niveluri de sistem).

Algoritmul de planificare favorizează threadurile cu prioritate mai mare, diverse *politici de planificare* trebuie utilizate doar pentru eficientizarea unui anumit algoritm, care însă trebuie să se poată executa corect şi în absenţa acestuia.

Principalele câmpuri şi metode ale clasei Thread sunt:

- *void start()* - lansează în execuţie noul thread, moment în care execuţia programului este controlată de cel puţin două threaduri: threadul curent ce execută metoda start şi noul thread ale cărui instrucţiuni sunt definite în metoda run ().

- *void run()* – defineşte corpul threadului nou creat, întreaga activitate a threadului va fi descrisă prin suprascrierea acestei metode
- *static void sleep ()* – pune în aşteptare threadul curent pentru un anumit interval de timp (*msecs*)
- *void join ()* - se aşteaptă ca obiectul thread ce apelează această metodă să se termine
- *suspend()* - suspendare temporară a threadului (*resume() este metoda* dual ce relansează un thread suspendat (implementările JDK ulterioare versiunii 1.2 au renunţat la utilizarea lor)
- *yield()* - realizează cedarea controlului de la obiectul thread, planificatorului JVM pentru a permite unui alt thread să ruleze
- *void interrupt()* – trimite o întrerupere obiectului thread ce o invocă (setează un flag de întrerupere a threadului activ). Metodele isInterrupted() şi interrupted() permit testarea stării de întrerupere a threadului apelant. Metoda interrupted() modifcă starea threadului curent (la un apel secund al ei starea threadului revine la cea iniţială).
- *static boolean interrupted()* - metodă statică, testează dacă threadul curent a fost întrerupt, resetează starea interrupted a threadului current
- *boolean isInterrupted* () - testează dacă un thread a fost întrerupt fără a modifica starea threadului
- *static Thread current Thread()* - returnează obiectul reprezentând threadul curent în execuţie
- *boolean isAlive()* - permite identificarea stării obiectului thread, astfel metoda returnează *true* dacă threadul a fost pornit şi nu a murit încă,respectiv *false* dacă threadul nu a fost pornit sau a murit, fără a putea diferenţia între un thread ce nu a fost încă pornit, respectiv unul ce a murit.
- *void SetDaemon(boolean on)* - apelată imediat înainte de start permite definirea threadului ca daemon. Un thread este numit daemon, dacă metoda lui run conţine un ciclu infinit, astfel încat acesta nu se va termina la terminarea threadului părinte.
- *getPriority()* - returnează prioritatea threadului curent
- *setPriority(newPriority)* -permite atribuirea pentru threadul curent a unei priorităţi dintr-un interval.

Metodele stop(), suspend() şi resume(),definite în versiuni anterioare au fost eliminate deoarece în cazul unei proiectări defectuoase a codului pot provoca blocarea acestuia (nu sunt thread-safe, altfel spus nu asigură execuţie corectă). O clasă aparte de thread-uri sunt cele Daemon care sunt thread-uri de serviciu (aflate în serviciul altor fire de execuţie). Când se porneşte maşina virtuală Java, există un singur fir de execuţie care nu este de tip Daemon şi care apelează metoda main(). JVM rămâne pornită cât există activ un thread care să nu fie de tipul Daemon. Metoda setDaemon poate fi utilizată pentru transformarea unui thread în daemon şi invers.

1.2.2.Controlul şi ciclul de viaţă al unui thread

Fiecare fir de execuţie are propriul său ciclu de viaţă: este creat, devine activ prin lansarea sa în execuţie şi la un moment dat, se termină. În continuare vom vedea mai îndeaproape stările în care se poate găsi un fir de execuţie. Diagrama din Figura 1.1. ilustrează aceste stări precum şi metodele care provoacă tranziţia dintr-o stare în alta. Aşadar, un fir de execuţie se poate găsi în una din următoarele stări ilustrate în Figura 1.1, acestea vor fi descrise în continuare.

Starea "New Thread" .
Un fir de execuţie se găseşte în această stare imediat după crearea sa, cu alte cuvinte după instanţierea unui obiect din clasa Thread sau dintr-o subclasă a sa.

```
Thread counterThread = new Thread ( this );
//counterThread se găseşte în starea New Thread
```

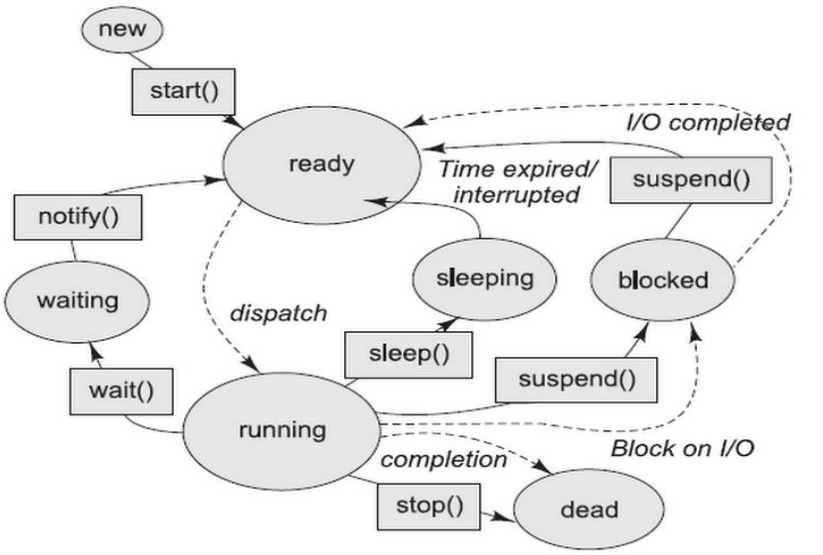


Figura 1.1. Stările unui thread Java [4]

În această stare firul de execuţie este "vid", el nu are alocate nici un fel de resurse sistem şi singura operaţiune pe care o putem executa asupra lui este lansarea în execuţie, prin metoda start(). Apelul oricărei alte metode în afară de start nu are nici un sens şi va provoca o excepţie de tipul IllegalThreadStateException.

Stări de tip "Runnable"(ready/running)

După apelul metodei start un fir de execuţie va trece în starea "Ready", adică se găseşte în execuţie.

```
counterThread.start();
//counterThread se găseşte în staarea ready
```

Metoda *start* realizează următoarele operaţiuni necesare rulării firului de execuţie:

- alocă resursele sistem necesare
- planifică firul de execuţie la CPU pentru a fi lansat
- apelează metoda run a obiectului reprezentat de firul de execuţie

Stări de tip "Not Runnable" (sleeping/blocked/waiting)

Un fir de execuţie ajunge în această stare în una din următoarele situaţii:

- este "adormit" prin apelul metodei *sleep.*
- a apelat metoda *wait,* aşteptând ca o anumită condiţie să fie satisfăcută
- este blocat într-o operaţie de intrare/ieşire

Metoda *sleep* este o metodă statică a clasei Thread care provoacă o pauză în timpul rulării firului curent aflat în execuţie, cu alte cuvinte îl "adoarme" pentru un timp specificat. Lungimea acestei pauze este specificată în milisecunde/ nanosecunde.

```
public static void sleep( long millis ) throws InterruptedException
public static void sleep( long millis, int nanos ) throws InterruptedException
```

Întrucât poate provoca excepţii de tipul InterruptedException apelul acestei metode se face într-un bloc de tip try-catch:

```
try { Thread.sleep(1000);
                    //face pauza de o secunda
                            } catch (InterruptedException e) {
                                    . . . }
```

După expirarea acestui interval threadul revine în starea Running, iar dacă procesorul este în continuare disponibil, îşi continua execuţia. Pentru fiecare tip de intrare în starea "Not Runnable", există o secvenţă specifică de ieşire din starea respectivă, care readuce firul de execuţie în stări Runnable. Acestea sunt:

- dacă un fir de execuţie a fost "adormit", atunci el devine Runnable doar după scurgerea intervalului de timp specificat de instrucţiunea sleep.
- dacă un fir de execuţie aşteaptă o anumită condiţie, atunci un alt obiect trebuie să îl informeze dacă acea condiţie este îndeplinită sau nu; acest lucru se realizează prin instrucţiunile *notify sau notifyAll*
- dacă un fir de execuţie este blocat într-o operaţiune de intrare/ieşire atunci el redevine Running atunci când acea operaţiune s-a terminat.

Starea "Dead"

Este starea în care ajunge un fir de execuţie la terminarea sa. Un fir de execuţie nu poate fi oprit din program printr-o anumită metodă, ci trebuie să se termine în mod natural la terminarea metodei run pe care o execută. Spre deosebire de versiunile curente ale limbajului Java, în versiunile mai vechi exista metoda stop a clasei Thread care termină forţat un fir de execuţie, însă ea a fost eliminată din motive de securitate.

1.2.3.Operaţii cu threaduri Java

a. Crearea şi pornirea unui thread

Există două tehnici pentru crearea unui nou thread de execuţie
şi implementarea metodei run() şi anume: extinderea clasei Thread (implementarea unei clase derivate) şi suprascrierea metodei run(), respective, definirea unei clase ce implementează o interfaţă Runnable

Pentru a decide care din cele două metode este necesară unui anumit context de programare este utilă următoarea observaţie: dacă clasa este derivată din altă clasă este de preferat să implementeze interfaţa Runnable (datorită lipsei mecanismului de moştenire multiplă în limbajul Java), iar în caz contrar, se utilizează extinderea clasei Thread, ambele soluţii fiind implementate în pachetul java.lang.

Varianta1.Extinderea clasei java.lang.Thread

Etapele de creare a threadului folosind subclasarea sunt :

1. se creează o clasă derivată din clasa Thread
2. se suprascrie metoda public void run() moştenită din clasa Thread
3. se instanţiază un obiect thread folosind new
4. se porneşte thread-ul instanţiat, prin apelul metodei start() moştenită din clasa Thread. Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program necesar unui thread după care să apeleze metoda run().

```
public class Fir
{ public static void main(String args[])
  {
     FirdeExecutie fir=new FirdeExecutie();
     fir.start();
     System.out.println("Revenim la main"); }}

class FirdeExecutie extends Thread
{
   public void run()
   {
      for(int i=0;i<10;i++)
         System.out.println("Pasul "+i);
      System.out.println("Run s-a terminat");}}
```

Variata2.Implementare folosind interfaţa java.lang.Runnable.

Pentru această metodă de implementare, etapele specifice sunt:

1. se creează o clasă care implementează interfaţa Runnable
2. se implemetează metoda run() din interfaţă
3. se instanţiază un obiect al clasei folosind new()
4. se creează un obiect din clasa Thread folosind un constructor care are ca parametru un obiect de tip Runnable (un obiect al clasei ce implementează interfaţa)
5. se porneşte thread-ul creat la pasul anterior prin apelul metodei start().

```
public class Fir
{
   public static void main(String args[])
  {
     FirdeExecutie fir=new FirdeExecutie();
     Thread thread=new Thread(fir);
     thread.start();
     System.out.println("Revenim la main");}}
 class A  {
  public void afis()
 {
   System.out.println("Este un exemplu simplu"); } }
class FirdeExecutie extends A implements Runnable
{
   public void run()
   {
      for(int i=0;i<5;i++)
         System.out.println("Pasul "+i);
           afis();
         System.out.println("Run s-a terminat");}}
```

Aceasta este o modalitate extrem de utilă atunci când clasa de tip Thread care se doreşte a fi implementată moşteneşte o altă clasă (Java nu permite moştenirea multiplă). Interfaţa Runnable descrie o singură metodă run().

Metoda start()creează resursele sistem necesare execuţiei threadului şi apelează metoda run. După revenire din această metodă, threadul se află în starea runnable, sistemul runtime Java implementează o schemă de planificare ce partajează procesorul (sau procesoarele) între toate threadurile active, astfel încât la un moment dat un thread aşteaptă accesul planificat la resursele sistem CPU.

Paralelismul real pentru sisteme multiprocesor pe care se execută un program multithread depinde de modul în care procesoarele sunt alocate threadurilor. Pentru un thread metoda *start ()* se poate apela o singură dată. Se poate pune întrebarea de ce sunt necesare două metode de creare a threadurilor. Argumentele sunt următoarele: dacă programul necesită control asupra cicluluid eviaţă a threadului , extinderea clasei Thread constituie abordarea acorectă, iar dacă este necesară mai multă flexibilitate în exinderea altor clase se va prefera implementarea interfeței Runnable.

b. Transformarea threadului în Not Runnable

Threadul devine *Not Runnable* în situaţia în care apare unul din următoarele evenimente:

- este invocată metoda *sleep(),* astfel deşi procesorul este disponibil threadul este pus în aşteptare un anumit interval de timp, dacă threadul revine în starea Runnable şi procesorul redevine disponibil threadul îşi va reîncepe execuţia.
- threadul apelează metoda wait() pentru a aştepta satisfacerea unei anumite condiţii, iar pentru a deveni Runnable, un alt thread trebuie să notifice threadul în aşteptare despre o modificare a condiţiei apelând *notify () sau notifyAll().*
- threadul e blocat la o operaţie I/O şi devine Runnable la terminarea operaţiei.

c.Terminarea unui thread

JVM va executa un thread cât şi threadurile create de acesta până la întâlnirea uneia din situaţiile: apelul metodei *exit()* a clasei Runtime iar managerul de securitate a permis îndeplinirea cu succes a acestui apel, respectiv până s-au terminat toate threadurile ce nu sunt daemon, prin revenirea metodei *run(),* apelul metodei *stop()*, respectiv recepţia unei întreruperi *interrupt()* din partea altui thread. După cum am văzut, un fir de execuţie nu poate fi terminat forţat de către program ci trebuie să-şi programeze singur terminarea sa. Acest lucru poate fi realizat în două modalităţi :

1.**prin scrierea unor metode run care să-şi termine execuţia** în mod natural; la terminarea metodei run se va termina automat şi firul de execuţie, acesta intrând în starea Dead.

2.**prin folosirea unei variabile de terminare.** În cazul când metoda run trebuie să execute o buclă infinită atunci aceasta trebuie controlată şi printr-o variabilă care să oprească această buclă atunci când dorim ca firul de execuţie să se termine. Uzual, aceasta este o variabilă membră a clasei care descrie firul de execuţie care fie este publică, fie este asociată cu o metoda care îi schimbă valoarea.

Metoda run() nu se termină natural, ea rulează la infinit aşteptând să fie terminată forţat folosind de exemplu metoda stop. Această metodă este însă "învechită" (deprecated) iar la compilarea programului vom obţine un mesaj de avertizare în acest sens. Putem evita metoda stop prin folosirea unei variabile de terminare.

Metoda *isAlive* () este folosită pentru a vedea dacă un fir de execuţie a fost pornit şi nu s-a terminat încă. Metoda returnează:

- true - dacă firul este în una din stările Runnable sau Not Runnable
- false - dacă firul este în una din stările New Thread sau Dead

Între stările Runnable sau Not Runnable, respectiv New Thread sau Dead nu se poate face nici o diferenţiere.

Nu este necesară distrugerea explicită a unui fir de execuţie,sistemul se ocupa de acest lucru. El poate fi fortat să dezaloce resursele alocate unui thread prin atribuirea cu null a variabilei care referea instanţa firului de execuţie: myThread = null .

Aşteptarea terminării unui thread. Dacă este necesară aşteptarea terminării unui thread se poate utiliza metoda *join()* a obiectului thread după care se aşteaptă, funcţie ce are acelaşi efect ca şi combinaţia *sleep()* şi *isAlive().* Apelul this.join() referă o aşteptare perpetuă, astfel threadul curent nu se va termina. Funcţia *join()* pentru un thread ce nu şi/a început execuţia sau deja s-a terminat va returna fără să aştepte.

d.Întreruperea unui thread

Un thread se termină la revenirea din metoda run(). Funcţia interrupt () solicită unui thread abandonarea activităţilor sale (din starea ready). Dacă threadul este blocat *(în stările wait* sau *sleep*) nu se poate autoinspecta pentru întrerupere. Dacă threadul este în aşteptare, va genera o excepţie Interrupted Exception (flagul corespunzător nu este setat), iar dacă threadul este blocat prin sincronizare, nu va genera excepţie, însă flagul corespunzător va fi setat.

Preemptiv vs. non-preemptiv **.**Sistemul runtime Java nu va întrerupe threadul curent pentru un alt thread de aceeaşi prioritate, este posibil însă ca sistemul de operare să implementeze propria politică de preempţie (întrerupere) a threadurilor. Un thread se consideră că se manifestă corect dacă este capabil să elibereze periodic CPU altor threaduri. Dacă acesta nu se poate auto-suspenda prin aşteptare la o condiţie, prin apelul metodelor *sleep* sau invocarea unor operaţii de intrare-ieşire, este necesară invocarea metodei *yield().* Prin această metodă, threadul renunţă de bună voie la serviciile procesorului şi maşina virtuală Java este anunţată că poate preda controlul unui alt thread pe care planificatorul decide să îl treacă în starea ready. În orice moment este posibilă verificarea execuţiei unui thread folosind funcţia *isAlive().*

e. Planificarea threadurilor

Execuţia într-o anumită ordine a mai multor fire de execuţie pe un singur procesor se numeşte *planificare (engl. scheduling)*. Sistemul Java de execuţie a programelor implementează un algoritm simplu, determinist de planificare, cunoscut sub numele de *planificare cu prioritati fixate*. Fiecare fir de execuţie Java primeşte la crearea sa o anumită prioritate. O prioritate este de fapt un număr întreg cu valori cuprinse între MIN_PRIORITY şi MAX_PRIORITY. Implicit prioritatea unui fir de execuţie nou creat are valoarea NORM_PRIORITY. Aceste trei constante sunt definite în clasa Thread:

public static final int MIN_PRIORITY - prioritatea minimă
public static final int MAX_PRIORITY - prioritatea maximă
public static final int NORM_PRIORITY – prioritatea implicită

Schimbarea ulterioară a priorităţii unui fir de execuţie se realizează cu metoda setPriority() a clasei Thread. Planificatorul Java lucrează în modul următor: dacă la un moment dat sunt mai multe fire de execuţie în starea Runnable, adică sunt pregătite pentru a fi executate, planificatorul îl va alege pe cel cu prioritatea cea mai mare pentru a-l executa. Doar când firul de execuţie cu prioritate maximă se termină sau este suspendat din diverse motive va fi ales un fir de execuţie cu o prioritate mai mică. În cazul în care toate firele au aceeaşi prioritate ele sunt alese după un algoritm simplu de tip "round-robin". De asemenea, planificarea este complet *preemptivă*: dacă un fir cu prioritate mai mare decât firul care se execută la un moment dat solicită procesorul, atunci firul cu prioritate mai mare este imediat trecut în execuţie ,iar celălalt trecut în aşteptare.

Planificatorul Java nu va întrerupe un fir de execuţie în favoarea altuia de aceeaşi prioritate, însă acest lucru îl poate face sistemul de operare în cazul în care acesta alocă procesorul în cuante de timp. Aşadar, un fir de execuţie cedează procesorul în una din situaţiile:

- un alt fir de execuţie cu o prioritate mai mare solicită procesorul
- metoda sa run se termină
- vrea să facă explicit acest lucru apelând metoda yield
- timpul alocat pentru execuţia sa a expirat

În nici un caz corectitudinea unui program nu trebuie să se bazeze pe mecanismul de planificare a firelor de execuţie, deoarece acesta poate fi imprevizibil şi depinde de la un sistem de operare la altul. Un fir de execuţie de lungă durată şi care nu cedează explicit procesorul la anumite intervale de timp astfel încât să poată fi executate şi celelalte fire de execuţie se numeşte fir de execuţie egoist şi trebuie evitată utilizarea lor, întrucât acaparează pe termen nedefinit procesorul, blocând efectiv execuţia celorlalte fire de execuţie până la terminarea sa.

f. Grupuri de threaduri

Gruparea firelor de execuţie pune la dispoziţie un mecanism pentru manipularea acestora ca un tot şi nu individual,astfel putem să pornim sau să suspendăm toate firele dintr-un grup cu un singur apel de metodă. Gruparea firelor de execuţie se realizează prin intermediul clasei *ThreadGroup.*

Fiecare fir de execuţie Java este membru al unui grup, indiferent dacă specificăm explicit acest lucru. Afilierea unui fir de execuţie la un anumit grup se realizează la crearea sa şi devine permanentă, în sensul că nu vom putea muta un fir de execuţie dintr-un grup în altul, după ce acesta a fost creat. În cazul în care creăm un fir de execuţie fără a specifica în constructor din ce grup face parte, el va fi plasat automat în acelaşi grup cu firul de execuţie care l-a creat. La pornirea unui program se creează automat un obiect de tip ThreadGroup cu numele *main*, care va reprezenta grupul tuturor firelor de execuţie create direct din program şi care nu au fost ataşate explicit altui grup. Cu alte cuvinte, putem să ignoram complet plasarea firelor de execuţie în grupuri şi să lăsam sistemul să se ocupe cu aceasta.

Exista situaţii când programul creează multe fire de execuţie, iar gruparea lor poate simplifica substanţial manevrarea lor. Un grup se compune dintr-o mulţime de threaduri şi poate conţine alte grupuri, formând un arbore, un thread poate accesa doar informaţia referitoare la propriul grup fără a avea acces la alte grupuri sau la grupul părinte. Orice thread aparţine unui grup specificat ca un parametru constructor sau moştenit. Threadurile unui grup pot fi enumerate, pentru un grup poate fi setată prioritatea maximă respectiv grupul de threaduri poate fi întrerupt din execuţie. Grupurile de threaduri sunt folosite deoarece permit manipularea mai multor threaduri la un singur apel de funcţie şi oferă bazele mecanismului de securitate folosit pentru lucrul cu threaduri. În mod implicit toate threadurile create în program aparţin unui grup definit de maşina virtuală Java, însă din motive de securitate doar programele standalone pot opera cu grupuri de threaduri.

Unele dintre metodele de manipulare a grupurilor de threaduri sunt preluate de la clasa Thread, prin extinderea semnificaţiei pentru grupuri. Un constructor de thread, în forma cea mai generală ataşează noul thread unui grup de threaduri, dacă argumentul lipseşte atunci este ataşat grupului curent. Grupul unui thread este stabilit la crearea acestuia şi nu mai poate fi modificat pe parcursul execuţiei sale, la terminare threadul este automat eliminat din grup Pentru a afla cărui grup aparţine un anumit fir de execuţie putem folosi metoda getThreadGroup a clasei Thread. Un grup poate avea ca părinte un alt grup, ceea ce înseamnă că firele de execuţie pot fi plasate într-o ierarhie de grupuri, în care rădăcina este grupul implicit main.

g.Comunicarea prin fluxuri de tip "pipe"

O modalitate deosebit de utilă prin care două fire de execuţie pot comunica este realizată prin intermediul *canalelor de comunicatii (pipes)*. Acestea sunt implementate prin fluxuri descrise de clasele
PipedReader, PipedWriter - pentru caractere, respectiv
PipedOutputStream, PipedInputStream - pentru octeţi
Constructorii acestor clase sunt:

```
public PipedReader( )
public PipedReader( PipedWriter pw ) throws IOException  public PipedWriter( )
public PipedWriter(PipedReader pr) throws IOException
```

În cazul în care este folosit constructorul fără argument conectarea unui flux de intrare cu un flux de ieşire se face prin metoda **connect**:

```
public void connect(PipedWriter pw) throws IOException
public void connect(PipedReader pr) throws IOException
```

Întrucât ,fluxurile care sunt conectate printr-un pipe trebuie să execute simultan operaţii de scriere/citire folosirea lor se va face în cadrul unor fire de execuţie. Funcţionarea obiectelor care instanţiază PipedWriter şi PipedReader este asemănătoare cu a canalelor UNIX (pipes). Fiecare capăt al unui canal este utilizat dintr-un fir de execuţie separat. La un capăt al pipeline-ului se scriu caractere, la celalalt se citesc. La citire, dacă nu sunt date disponibile firul de execuţie se va bloca. Se observă că acesta este un comportament tipic producător-consumator, firele de execuţie comunicând printr-un canal. Realizarea conexiunii se face astfel:

```
PipedWriter pw1 = new PipedWriter();
PipedReader pr1 = new PipedReader(pw1);
sau
PipedReader pr2 = new PipedReader();
PipedWriter pw2 = new PipedWriter(pr2);
sau
PipedReader pr = new PipedReader();
PipedWriter pw = new PipedWriter();
pr.connect(pw) //echivalent cu
pw.connect(pr);
```

Scrierea şi citirea pe/de pe canale se realizează prin metodele uzuale read şi write în toate formele lor.

Exemplu de comunicare pipe:

Thread Scriitor

```
import java.io.*;
class FirScriitor extends Thread{private PipedOutputStream po;
FirScriitor(){po = new PipedOutputStream();}
public void run()
{try{while (true)
       {int d = (int)(10*Math.random());
           System.out.println("Fir scriitor trimite : "+d);
              po.write(d);
              sleep(400);}
       }catch(Exception e){}}
PipedOutputStream getPipe(){return po;}}
```

Thread Cititor

```
import java.io.*;
class FirCititor extends Thread
{private PipedInputStream pi;
FirCititor()
{pi = new PipedInputStream();     }
public void run()
{try
       {while (true)
       {if (pi.available()>0)
              {System.out.println("Fir cititor a primit : "+pi.read());}}
       }catch(Exception e){}}
void conect(PipedOutputStream os)throws Exception
{pi.connect(os);}}
```

Comunicarea Scriitor-Cititor

```
public class Test
{public static void main(String args[])
       {FirCititor fc = new FirCititor();
              FirScriitor fs = new FirScriitor();
              try
              {fc.conect(fs.getPipe());
                     fc.start();
                     fs.start();
              }catch(Exception e)
              {e.printStackTrace();}}}
```

1.2.4.Sincronizarea threadurilor

În secţiuena precedent s-a prezentatcum pot fi create fire de execuţie *independente şi asincrone,* care nu depind în nici un fel de execuţia sau de rezultatele altor fire de execuţie. Există însă numeroase situaţii când fire de execuţie separate, dar care rulează concurent, trebuie să comunice între ele pentru a accesa diferite resurse comune sau pentru a-şi transmite dinamic rezultatele "muncii" lor, operând astfel în manieră cooperativă.

Mecanismul monitor Java. Dacă mai multe threaduri operează simultan asupra unui obiect, datele acestuia pot fi corupte. Deoarece comutarea de context între threaduri poate surveni în orice moment, prevenirea accesului concurent este necesară chiar dacă sistemul este uniprocesor. Blocul de cod ce accesează acelaşi obiect din două threaduri separate este numit *secţiune critică*. Pentru a permite unui singur thread accesul la secţiunea critică, este necesar un mecanism de excludere mutuală. Excluderea mutuală presupune utilizarea unor algoritmi, implementarea de semafoare, respectiv monitoare. Mecanismul de sincronizare de bază în Java este *monitorul*. În Java orice obiect este un potenţial monitor având asociată o coadă de aşteptare şi mecanisme de semnalizare.

Monitorul încapsulează *datele* obiectului partajat, *procedurile de acces sincronizat* la aceste obiecte şi un *constructor* de iniţializare a obiectului monitor. Obiectele monitor aplică principiul excluderii mutuale pentru grupul de proceduri sincronizate, astfel *accesul sincronizat presupune serializarea threadurilor.*

Referitor la utilizarea metodelor-secţiunilor de program sincronizate cât şi pentru a evita condiţiile de tip "cursă" în accesul la date, asigurând astfel integritatea acestora se impun următoarele precizări:

- *secţiune critică* poate reprezenta o *metodă* sau *un bloc* de cod şi este identificată de cuvântul cheie synchronized. În primul caz mecanismul de sincronizare este asociat obiectului curent (monitorului său), iar în cel de al doilea se poate alege obiectul de sincronizare ce poate fi un obiect static sau cel curent, evitând alegerea unui obiect a cărui valoare este modificată în cadrul blocului
- Java asociază un lacăt (monitor) oricărui (any) obiect, acapararea şi eliberarea lacătului este realizată în mod *automat şi atomic* de către sistemul runtime la intrarea, respectiv ieşirea din blocul de cod declarat *sincronizat.*
- modificările efectuate într-o metodă sincronizată devin *vizibile* celorlalte metode sincronizate referitor la acelaşi obiect
- apelul unei metode sincronizate determină ca threadul ce a apelat metoda să plaseze un *lacăt* asupra obiectului a cărui metodă a fost apelată
- alte threaduri nu pot executa o metodă sincronizată asupra unui obiect pană la eliberarea lacătului, acestea vor fi blocate.
- atunci când threadul ce deţine lacătul va reveni din metoda sincronizată lacătul obiectului va fi în mod automat eliberat
- unul din threadurile ce aşteaptă acapararea lacătului îl va prelua şi va apela metoda sincronizată.
- citirile şi scrierile unor câmpuri volatile sunt atomice şi vizibile altor câmpuri

Exemplu de declarare a metodelor sincronizate pentru un cont bancar:

```
class Account {
private double balance;
public Account(double initialDeposit) { balance = initialDeposit;   }
public synchronized double getBalance() {  return balance;   }
public synchronized void deposit(double amount) { balance += amount;   }  }
```

- declararea metodei ca synchronized presupune ca lacătul obiectului curent (this) să fie acaparat înaintea execuţiei metodei
- mecanismul de sincronizare a blocurilor de cod, permite specificarea explicită a lacătului unui anumit obiect ce trebuie acaparat de un thread înaintea execuţiei blocului respectiv (poate fi un obiect Java de tip any, sau este posibil să fie un obiect ce nu este utilizat în cadrul blocului sincronizat)
- creşterea gradului de paralelism poate fi implementată prin declararea unui întreg bloc de cod ca synchronized.

Mecanismul wait-notify. Un thread ce deţine exclusivitatea asupra unui obiect de tip monitor poate ceda temporar exclusivitatea accesului. Apelul metodei *wait(),* are drept rezultat pentru threadul curent oprirea temporară a execuţiei şi trecerea în starea *waiting(),* iar reluarea accesului exclusiv are loc atunci când threadul se trezeşte ca urmare a unui mesaj de anunţ generat de un alt thread activ ce va executa una din metodele *notify()* sau *notifyAll().* În urma acestui eveniment threadul recapătă exclusivitatea cedată şi continuă execuţia oprită anterior după apelul *wait().* Un obiect posedă două cozi asociate şi anume :

- o coadă pentru *excluderea prin sincronizare* a altor threaduri
- o coadă pentru *aşteptare la condiţie*, însă se poate afla la un moment dat doar într-una din cele două cozi.

Anunţul (notificarea) efectuat cu metoda *notify()* este adresat tuturor threadurilor aflate în starea Waiting, dar unul singur îşi va relua activitatea, iar pentru un anunţ efectuat cu *notifyAll()* toate threadurile în aşteptare îşi vor relua activitatea. Este posibil ca un thread să solicite intrarea sa în starea Waiting pentru un interval de timp bine delimitat deoarece metoda *wait ()* permite specificarea duratei maxime de aşteptare. Dacă un alt thread apelează metoda *interrupt()* a threadului în aşteptare, execuţia acestuia este reluată cu secvenţa *catch* ce interceptează întreruperea.

Cele două metode *wait()- notify()* trebuie apelate *în interiorul unei metode sau bloc* ***sincronizat.*** Tehnica integrării acestui mecanism cu metodele de sincronizare este similară metodelor de sincronizare de nivel jos, caz în care variabilele condiţionale au asociate variabile tip mutex pe care le blochează-deblochează funcţie de faza aşteptării. Astfel în Java înaintea intrării în aşteptare prin wait(),threadul curent eliberează obiectul de sincronizare blocat şi îl obţine din nou când threadul este anunţat că a fost realizată condiţia aşteptată. Pentru o utilizare corectă a acestui mecanism sunt necesare următoarele precizări :

- dacă *notify()*, notifyAll() este apelată dintr-un thread fără a exista un thread care să aştepte, se va reveni fără eroare
- metoda *wait ()* eliberează obiectul de sincronizare la intrarea în aşteptare şi îl reobţine la recepţionarea notificării
- dacă există mai multe threaduri ce aşteaptă o anumită condiţie nu se poate şti care thread va primi notificarea, maşina virtuală Java şi mecanismul de planificare a threadurilor determină ordinea precisă
- dacă se doreşte apelul funcţiilor wait()-notify() din metode statice este necesară declararea în cadrul clasei a unui obiect de sincronizare static pentru care vor fi apelate aceste funcţii
- diferenţa între metodele wait() şi sleep() constă în aceea că metoda sleep *nu necesită apelul obligatoriu dintr-un bloc sincronizat* şi nu implică blocarea sau deblocarea unui obiect de sincronizare.
- *NotifyAll* nu eliberează lacătul
- nici un thread nou nu poate intra în coada wait asociată pe durata cât threadul notificator deţine încă lacătul. Funcţia va transfera toate threadurile aflate în aşteptare din coada de aşteptare în coada de blocare aşteptând momentul în care pot continua.

Evitarea metodelor stop() şi suspend(). Metodele *stop, resume şi suspend()* lasă obiectele în stări inconsistente. Un thread suspendat forţat poate deţine infinit un lacăt, iar dacă va apela ulterior resume() va genera blocaj datorat aceluiaşi lacăt. Threadul trebuie să verifice dacă a fost intrerupt. Versiuni sigure pentru operaţiile suspend() şi resume() pot fi implementate bazat pe operaţiile *wait() şi notifyAll(),* astfel threadul va verifica cererea de suspendare şi va controla momentul în care îşi poate întrerupe în siguranţă activităţile.

Detectarea şi evitarea blocării în programele concurente este realizată prin proiectarea corectă a algoritmilor. Blocajele sunt generate de dependenţele circulare între threaduri şi lacătele pe care acestea încearcă să le acapareze. O definire simplă a blocajului este următoarea: un thread deţine un lacăt (resursă) necesar unui alt thread, el la rândul său încercând să obţină un lacăt deţinut de un alt thread.Evitarea blocajului este dificilă deoarece presupune interacţiuni corecte între obiecte multiple cooperante.

Următorul set de reguli permit evitarea blocajelor în secvenţele de cod ce necesită concurenţă:

- evitarea apelurilor blocante (operaţii de intrare-ieşire) în cadrul unui bloc sincronizat
- evitarea apelurilor imbricate sincronizate
- structurarea liniară a fluxului de control prin aplicarea unor strategii şi/sau euristici între care reordonarea resurselor, creşterea granularităţii (obţinerea prealabilă a lacătului exterior)
- utilizarea unor mecanisme de *tip timeout –retry* (poate genera blocaje de tip livelock)
- implementarea unor mecanisme de tip *transfer de jeton* ce permit evitarea unor sincronizări dinamice a accesului la obiectul resursă
- *planificare explicită* a threadurilor
- modificări de semantică a cerinţelor de atomicitate totală (ex. propagarea modificărilor şi actualizarea datelor nu necesită tratare atomică)

1.3. Exemple

Se vor testa exemplele ce ilustrează modul de utilizare a API-ului Java multithreading pentru următoarele mecanisme specifice : utilizare diverse metode ale clasei thread, blocuri sincronizate, prioritizare și planificare threaduri, exemple pe care le puteți descărca de la adresa *https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/01_JavaMT*

1.4. Întrebări teoretice

1.4.1. Definiți conceptul de thread. Explicați succint diferențe conceptuale și de utilizare între proces și thread.

1.4.2. Explicați conceptul de comutare de context.

1.4.3. Care sunt avantajele multitaskingului realizat prin intermediul threadurilor vs al proceselor.

1.4.4. Descrieți ciclul de viață al unui thread, identificând stările și tranzițiile dintre acestea.

1.4.5. Explicați conceptul de prioritate prin intermediul unui exemplu.Cum se asigură prioritizarea threadurilor ?

1.4.6. Descrieți modalitățile prin care pot fi create threaduri și identificați cazurile de aplicare.

1.5. Probleme propuse

1.5.1. Să se implementeze problema *producător – consumator* utilizând două mecanisme de sincronizare diferite. O primă soluţie poate fi considerată utilizarea unei cozi intermediare, se cunoaşte faptul că producătorul necesită 1000 ms pentru a produce o valoare, respectiv consumatorul necesită 2000 ms pentru a o consuma.

1.5.2. Să se implementeze folosind facilităţile de sincronizare specifice limbajului Java problema accesului concurent a scriitorilor şi cititorilor la un fisier. Realizaţi o soluţie ce oferă acces *concurent* pentru citire şi exclusiv pentru scriere.

1.5.3. Intr-un birou sunt 8 functionari care din când în când tipăresc la imprimantă documente, nu toți elaborează documentele în acelaşi ritm. Fiindcă au o singură imprimantă în birou, poate tipări doar o singura persoană la un moment dat. Să se simuleze functionarea biroului.

1.5.4. Scrieţi un program Java care generează două fire de execuţie pentru parcurgerea unui String de la cele două capete. Folosiţi doi pointeri a căror valoare se incrementează, respectiv se decrementează într-o funcţie din memoria comună (în memoria comună se află String-ul.)

1.6.Referințe bibliografice

1. Tutorial Oracle https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
2. Tutorial cu exemple https://www.tutorialspoint.com/java/java_multithreading.htm
3. Tuli S, Multithreading in Java https://dzone.com/articles/java-thread-tutorial-creating-threads-and-multithr
4. Buyya R., http://www.buyya.com/java/Chapter14.pdf

2. Java Concurrency - mecanisme pentru paralelism și concurență

Versiunea JDK 5.0 a constituit un pas major în programarea concurentă, astfel mașina virtuală Java a fost îmbunătățită semnificativ pentru a permite claselor să profite de suportul pentru concurență oferit la nivel hardware prin sisteme multicore. Deasemenea, un set bogat de noi clase au fost adăugate pentru a face mai ușoară dezvoltarea de aplicații concurente. Pachetul *java.util.concurent* aduce un set bine testat și foarte performant de funcții și structuri de date pentru *concurență și paralelism.*

2.1.Obiective

- Studiul structurilor de date și mecanismelor de programare concurentă /paralelă moderne din Java Concurrency
- Implementarea unor aplicații ilustrative cu aceste funcții și structuri de date și analiza performanței.

2.2.Concepte

Îmbunătățirile aduse din perspectiva suportului pentru concurență pot fi structurate în 3 categorii:

a) Modificări la nivelul mașinii virtuale java
Procesoarele moderne oferă suport hardware pentru concurență, în forma unor instrucțiuni *compare-and-swap* (CAS). Înainte de JDK 5.0, singura primitivă în limbajul Java pentru coordonarea accesului între thread-uri era *sincronizarea (synchronized).* Prin expunerea CAS se oferă posibilitatea dezvoltării unor clase Java foarte scalabile pentru aplicații ce solicită o astfel de abordare.

b) Clase utilitare de nivel scăzut
De exemplu clasa *ReentrantLock* oferă funcționalitate asemănătoare cu soluția *synchronized*, dar cu un control mai bun asupra blocării (temporizare lacăte - timed locks, verificare lacăte - lock polling, etc.) și astfel o mai bună scalabilitate.

c) Clase utilitare la nivel înalt
Clase care implementează: monitoare, semafoare, lacăte, bariere, thread-pools și colecții de date cu acces thread-safe. Acestea sunt oferite dezvoltatorilor de aplicații pentru a construi diverse soluții.

2.2.1. Colecții de date pentru mecanismele de concurență

Înainte de JDK 5.0 principalul mecanism pentru asigurarea faptului că o porțiune de cod este thread-safety ,altfel spus accesibilă unui singur thread la un moment dat era primitiva *synchronized*. Pachetul *java.util.concurrent* oferă noi primitive pentru asigurarea acestei proprietăți, precum și o serie de clase utilitare care nu necesită sincronizare adițională în codul aplicației, simplificând astfel codul aplicației.

Framework-ul Collections este un framework flexibil pentru reprezentarea colecțiilor de obiecte, folosind interfețele de bază *Map, List, Set.* Câteva dintre implementări sunt Thread-Safe (*Hashtable, Vector*), celelalte pot fi transformate în thread-safe cu ajutorul colecțiilor, și anume *Collections.synchronizedMap(), Collections.synchronizedList(),Collections.synchronizedSet().*

Pachetul *java.util.concurrent* adaugă noi colecții concurente: *ConcurrentHashMap, CopyOnWriteArrayList* și *CopyOnWriteArraySet*. Scopul acestor clase este să îmbunătățească performanța și scalabilitatea oferită de tipurile de colecții de bază.

Iteratorii s-au schimbat de asemenea în JDK 5.0. Dacă până la versiunea 5.0 nu se permitea modificarea unei colecții în timpul iterației, noii iteratorii oferă o vedere consistentă asupra colecției, chiar dacă aceasta se schimbă în timpul iterării.

Deasemenea,,*CopyOnWriteArrayList* și *CopyOnWriteArraySet* sunt versiuni îmbunătățite ale structurilor Vector și ArrayList. Îmbunătațirile sunt aduse în special la nivelul iterației,astfel, dacă în timpul parcurgerii unui *Vector* sau a unui *ArrayList* colecția este modificată, se va arunca o excepție, iar noile clase rezolvă această problemă.
Se oferă două noi structuri și interfețe pentru utilizarea cozilor : *Queue* și *BlockingQueue*.

- *Cozi (Queue)* Există două implementări principale, care determină ordinea în care elementele unei cozi sunt accesate: *ConcurrentLinkedQueue* (acces FIFO) și *PriorityQueue* (acces pe bază de priorități).
- *Cozi cu blocare (BlockingQueue*) Acest tip de cozi sunt folosite atunci când se dorește blocarea unui thread, în situația în care anumite operații pe o coadă nu pot fi executate. Un exemplu ar fi cazul în care consumatorii extrag mai greu din coadă informația decât ea este plasată în coadă de producători. Prin folosirea *BlockingQueue* se blochează automat producătorii până când se eliberează un element din coadă. Implementările interfeței *BlockingQueue* sunt: *LinkedBlockingQueue, PriorityBlockingQueue, ArrayBlockingQueue si SynchronousQueue*.

2.2.2.Thread Pools și Framework-ul Executor

Un mecanism clasic pentru managementul unui grup mare de task-uri este combinarea unei cozi de lucru (work *queue)* cu un set de threaduri (*thread pool)*. Astfel, *work queue* este o coadă de taskuri ce trebuie procesate, iar un *thread pool* este o colecție de thread-uri care extrag sarcini din coada și le execută. Când un *thread worker* termină o sarcină, se întoarce la coadă pentru a vedea dacă mai există sarcini de executat, iar dacă da, extrage sarcina din coada și o execută.

Pachetul *java.util.concurrent* conține un întreg framework pentru manangementul execuției task-urilor care implementează *Runnable.* Un aspect foarte important al folosirii framework-ului *Executor* este faptul că se permite *decuplarea lansării task-urilor de politica de execuție a lor*. Aceasta permite schimbarea politicii de execuție foarte ușor, fără a fi nevoie de modificări majore în cod.

Interfata Executor este foarte simplă:

```
public interface Executor {
        void execute (Runnable command); }
```

Politica de execuție a task-urilor depinde de implementarea de *Executor* aleasă.
Clasa *Executors* oferă diverse metode statice pentru obținerea de instanțe de diferite implementări de executori.

```
Executors.newCachedThreadPool()
Executors.newFixedThreadPool(int n)
Executors.newSingleThreadExecutor()
```

Executorii returnați de primele două metode sunt instanțe ale clasei *ThreadPoolExecutor*, care poate fi intens customizată în funcție de necesități.

Interfața Future **.**Această interfață permite reprezentarea unui task care poate s-a terminat, care este în execuție sau care încă nu a început să fie executat. Prin intermediul interfeței *Future* se pot anula taskuri care încă nu s-au terminat de executat, se poate verifica dacă un task s-a încheiat sau a fost anulat și se poate aștepta după rezultatul returnat de un task. *FutureTask* este o implementare a interfeței *Future* și are constructori care permit execuție peste o instanță de *Runnable*. Deoarece și *FutureTask* implementează *Runnable*, task-ul obținut poate fi direct transmis către un *Executor*.

2.2.3. Clase pentru sincronizare

O altă categorie de clase folositoare adăugate în *java.util.concurrent* este categoria claselor de sincronizare. Aceste clase coordonează și controlează fluxul execuției pentru unul sau mai multe thread-uri. Exemple de clase de sincronizare: *Semaphore,Mutex, CyclicBarrier, CountdownLatch, și Exchanger* .

Semaphore - implementează un semafor clasic, care are un număr dat de permisii ce pot fi cerute și eliberate. Acesta este folosit pentru a restricționa numărul de thread-uri ce pot avea simultan acces concurent la o resursă.

Mutex - un caz special de semafor, cu o singură permisie (permite doar acces exclusiv).

CyclicBarrier- oferă un ajutor de sincronizare: permite unui set de thread-uri să aștepte ca întreg setul să ajungă la o barieră comună.

CountdownLatch - oarecum similar cu *CyclicBarrier* prin faptul că permite coordonarea unui grup de thread-uri. Diferența este că atunci când un thread ajunge la barieră, nu se blochează ci doar decrementează valoarea inițială a lacătului. Este util când o problemă este divizată între mai multe thread-uri, fiecare realizând, o parte. Când un thread termină de rezolvat, decrementează contorul. Thread-ul de control poate verifica dacă lacătul a ajuns la valoarea 0, ceea ce înseamnă că toate thread-urile au terminat execuția.

Exchanger- facilitează schimbul bidirecțional între două thread-uri care cooperează. Este similar unei bariere de tip *CyclicBarrier* cu o valoare inițială de 2, la care se adaugă posibilitatea ca thread-urile să facă schimb de informații când ajung la barieră.

2.2.4.Facilități de nivel scăzut

Limbajul Java are o facilitate integrată de blocare și anume specificatorul *synchronized*. Când un thread achiziționează un monitor, alte thread-uri se vor bloca dacă încearcă să achiziționeaze același monitor, până când primul thread eliberează monitorul. Sincronizarea asigură și că valorile variabilelor modificate de un thread sunt vizibile thread-urilor care preiau accesul aceluiași lock ulterior.

Interfața *Lock* este o generalizare a funcționalității oferite de synchronized, permițând diverse implementări care adaugă o serie de funcționalități noi cele mai importante fiind: *timed waits, interruptible waits, lock polling, multiple condition-wait sets per lock si non-block-structured locking*.

ReentrantLock. Este o implementare a interfeței *Lock*, mult mai scalabilă, cu utilizare recursivă. Oferă viteză îmbunătățită la gestiunea mai multor thread-uri care vor să acceseze aceași resursă.

Conditions. La fel cum interfața *Lock* este generalizarea pentru *synchronized*, interfața *Condition* este o generalizare a metodelor *wait()* si *notify()* din clasa *Object*.

Variabile atomice. Deși sunt foarte rar folosite direct de utilizatori, unele dintre cele mai semnificative clase nou introduse, sunt clasele pentru variabile atomice: (*AtomicInteger, AtomicLong, AtomicRefer*, etc.). Aceste clase expun îmbunătățirile aduse mașinii virtuale, permitand operații atomice de citire-scriere. Aproape toate clasele din *java.util.concurrent* sunt construite peste *ReentrantLock* care la

rândul ei este construită utilizând clasele *atomic-variable*. Aşadar, deşi sunt transparente pentru majoritatea utilizatorilor, această categorie de clase este cea care aduce principalele îmbunătăţiri de scalabilitate oferite de *java.util.concurrent.*

2.2.5. Java Concurrency Framework

Java Concurrency Framework este un framework ce oferă o serie de servicii care permit utilizarea programării concurente în aplicaţiile moderne.

Cele mai importante pachete pe care acesta le pune la dispoziţie sunt:

- *java.util.concurrent.atomic* - ce oferă o mai mare flexibilitate în utilizarea de lacăte şi condiţii, în detrimentul sintaxei clasice
- *java.util.concurrent.locks* - clasele din acest pachet extind noţiunea de valoare volatilă, câmpuri şi şiruri de elemente cărora, de asemenea, le oferă operaţii atomice

a)Executori – java.util.concurrent.Executor

Atunci când este necesar să fie rulate mai multe sarcini complexe, în paralel şi să se aştepte finalizarea tuturor pentru ca mai apoi să se returneze o valoare, devine destul de dificilă conceperea unui cod bun care să le sincronizeze. Pentru a soluţiona această problemă, Java introduce Executor, o interfaţă ce permite crearea seturilor de thread-uri, susţine sincronizarea şi execuţia lor.

Interfaţa Executor declară o singură metodă - public void execute (Runnable obj). Pentru a o utiliza este nevoie de:

- implementarea interfeţei
- crearea unui obiect ce implementează interfaţa Runnable :Executor ex = new MyExecutor ();
- apelarea metodei executate ex. execute(unObiectRunnable);

```
import java.util.concurrent.*;
public class MyExecutor implements Executor {
 public void execute(Runnable obj) {
    obj.run(); // execuţia va avea loc în threadul apelant
       new Thread(obj).start(); // execuţia va avea loc într-un thread distinct }}
```

Dacă este nevoie de generarea şi gestionarea unui număr mai mare de thread-uri se poate crea un set de thread-uri (thread-pool). Un set de thread-uri poate fi reprezentat printr-o instanţă a clasei ExecutorService. Acesta poate fi de mai multe tipuri :

- *Single Thread Executor* – un set care conţine un singur thread; codul se va executa secvenţial
- *Fixed Thread Pool* – un set care conţine un număr fix de thread-uri; dacă un thread nu este disponibil pentru un task, acesta se pune într-o coadă şi aşteaptă finalizarea unui alt task
- *Cached Thread Pool* – un set care creează atâtea thread-uri cîte sunt necesare pentru executarea unui task în paralel
- *Scheduled Thread Pool* – un set creat pentru planificarea task-urilor viitoare
- *Single Thread Scheduled Pool* – un set care conţine un singur thread utilizat în planificarea task-urilor viitoare.

b)Thread Factory – java.util.concurrent.ThreadFactory Cu ajutorul *Thread Factory* se creează anumite tipuri de thread-uri într-un mod standardizat. Singura metodă disponibilă este n*ewThread* (Runnable r) care se suprascrie în clasele care o implementează, pentru crearea unor thread-uri customizate. Un exemplu de utilizare al acestei interfeţe îl gasiţi mai jos:

```
import java.util.concurrent.*;
public class MyThreadFactory implements ThreadFactory {
public Thread newThread(Runnable r) {
       return new Thread(r); }
```

```
public static void main(String[] args) {
            MyThreadFactory    mtf    =    new
       MyThreadFactory();
       Thread t = mtf.newThread(new MyThread());
            t.start(); } }

class MyThread extends Thread
{    public void run() {
            System.out.println("Inside: " + this.getName()); } }
```

Futures – java.util.concurrent.Future <V>

```
public class FutureTaskExample {
public static void main(String[] args) {
        MyCallable callable1 = new MyCallable(1000);
       MyCallable callable2 = new MyCallable(2000);
       FutureTask<String> futureTask1 = new FutureTask<String>(callable1);
       FutureTask<String> futureTask2 = new FutureTask<String>(callable2);
       ExecutorService executor = Executors.newFixedThreadPool(2);
       executor.execute(futureTask1);
       executor.execute(futureTask2);
       while(true) {
         try
         {if (futureTask1.isDone() && futureTask2.isDone())
               {System.out.println("Done");
               executor.shutdown();
               return; }
               if(!futureTask1.isDone())
               {System.out.println("FutureTask1 output= "+futureTask1.get());}
               System.out.println("Waiting for FutureTask2 to complete");
               String s = futureTask2.get(200L, TimeUnit.MILLISECONDS);
               if (s != null)
               {System.out.println("FutureTask2 output="+s);  }  }
         catch(InterruptedException | ExecutionException e)
         { e.printStackTrace();  }
         catch(TimeoutException e)
         {
       ……
         }}}}
```

Future reprezintă rezultatul unei acțiuni asincrone. Metodele oferite de această interfață se folosesc pentru a verifica dacă acțiunea *a luat sfârșit,* pentru a *aștepta terminarea* unei acțiuni și pentru a returna rezultatul obținut în urma execuției.

c)Cozi

Coada (FIFO – First In First Out) reprezintă o structură de date asemănătoare unei liste în care primul element introdus va fi și primul extras atunci când se execută o extragere. Începând cu Java 1.7, există mai multe tipuri de cozi implementate și care aparțin tot pachetului java.util.concurrent. Cele mai importante dintre acestea sunt: *AbstractQueue, ArrayBlockQueue, BlockingQueue, ConcurrentLinkedQueue, DelayQueue, e*tc.
Spre exemplu, coada de tipul *ArrayBlockingQueue* care implementează interfața *BlockingQueue* este utilizată în abordarea problemei ” producător-consumator”. În acest caz, coada are la bază o listă de dimensiune fixă și toate elementele cozii trebuie să fie de același tip. Constructorii clasei sunt:

- *ArrayBlockingQueue (int dim)* – creează o coadă care poate conține până la dim înregistrări
- *ArrayBlockingQueue (int dim, boolean fair)* – firele de execuție blocate sunt eliberate în ordinea blocării dacă fair = true și în mod aleator în caz contrar. Metodele acestei clase sunt:
    - *void put (o*) throws InterruptedException – introduce obiectul o în coadă, iar dacă operația nu este posibilă atunci thread-ul care a lansat-o este blocat
    - *boolean offer(o)* – introduce obiectul o în coadă și returnează true; dacă operația nu se poate efectua returnează false fără blocarea thread-ului care a lansat operația

- *peek()* – returnează elementul din vârful cozii
- *poll()* – returnează şi şterge elementul din vârful cozii; dacă nu mai există elemente în coadă returnează null
- *take()* throws InterruptedException - returnează şi şterge elementul din vârful cozii; dacă nu mai există elemente în coadă blochează thread-ul care a lansat operația
- *void clear()* - şterge conținutul cozii

Un exemplu de implementare a claselor Producer – Consumer devine cu aceste noi construcții:

```
public class Producer implements Runnable{
protected BlockingQueue queue = null;

public Producer(BlockingQueue queue) {
       this.queue = queue; }
@Override
public void run() {
       try {
              queue.put("1");
              Thread.sleep(1000);
              queue.put("2");
              Thread.sleep(1000);
              queue.put("3");
       } catch (InterruptedException e) {
              e.printStackTrace();}}}


public class Consumer implements Runnable{
       protected BlockingQueue queue = null;
       public Consumer(BlockingQueue queue)
       {this.queue=queue;}
       @Override
       public void run() {
       try {
              System.out.println(queue.take());
              System.out.println(queue.take());
              System.out.println(queue.take());
       } catch (InterruptedException e) {
              e.printStackTrace();}}}

public class BlockingQueueExample {
public static void main(String[] args) throws Exception {
      BlockingQueue queue = new ArrayBlockingQueue (1024);
      Producer producer = new Producer(queue);
      Consumer consumer = new Consumer(queue);
      new Thread(producer).start();
      new Thread(consumer).start();
      Thread.sleep(4000);}}
```

Programarea multithreading poate deveni uneori dificilă, mai ales când e necesară sincronizarea mai multor fire de execuție ce folosesc aceleaşi resurse simultan. Din acest motiv au fost introduse mai multe metode de sincronizare prezentate în cele ce urmează.

d) Excluderea reciprocă

Modificatorul *synchronized* ataşat unei metode permite unui singur fir de execuție să execute, la un moment dat, o singură metodă dintre cele cu acest modificator, astfel se asociază un mecanism de lacăt fiecărui obiect. Atunci când un thread apelează o metodă synchronized, lacătul se blochează şi nu permite accesul unui alt thread decât după eliberarea sa.

În acest fel se asigură excluderea reciprocă a firelor de execuție la resursele / metodele implicate. Modificatorul synchronized poate fi evitat prin utilizarea clasei *ReentrantLock()* ce implementează interfața *java.util.concurrent.Lock*. Aceasta defineşte următoarele metode:

- *void lock()* – închide lacătul; mai exact, orice thread care încearcă să apeleze o metodă și întâlnește lacătul blocat va fi suspendat.
- *void unlock()* – deschide lacătul
- *boolean tryLock()* – verifică dacă lacătul este blocat sau nu
- *Condition newCondition()* – creează o instanță de tip Condition atașată lacătului.

Sunt prezentate două moduri de incrementare a unei valori partajate de un număr de ori, prima folosind modificatorul synchronized, iar a doua utilizând Lock.

Metoda 1

```
public class Inc {
int n = 0;
void add()
{for (int i = 0; i < 100; i++)
{synchronized(this) n =n+1;}}}
```

Metoda 2

```
public  class
Inc {
int n = 0;
Lock l = new ReentrantLock();
void add()
{l.lock();
try {
for (int i = 0; i < 100; i++)
{n = n+1;}}
finally {
l.unlock();}}}
```

O altă modalitate de implementare a excluderii reciproce o reprezintă semafoarele, reprezentate în Java prin clasa *Semaphore.* Semafoarele se utilizează pentru a limita și controla accesul la o resursă partajată de mai multe threaduri. Înainte să obțină o resursă, un thread trebuie să obțină permisiunea de la semafor – adică validarea că resursa este disponibilă, apoi, când termină de utilizat resursa respectivă, thread-ul se întoarce la semafor pentru a semnala că aceasta este din nou disponibilă.

e)Sincronizarea prin barieră

Sincronizarea barieră presupune existența mai multor procese care execută aceeași secvență de cod care conține o instrucțiune cu rol de barieră. Poate fi vazută ca un set de thread-uri care se așteaptă reciproc să ajungă la un anumit punct comun, o barieră, după care se deblochează și își continuă execuția. Clasa *CyclicBarrier* permite programarea simplă a acestui tip de sincronizare. Se numește ciclic deoarece poate fi reutilizată după ce s-a ajuns la punctul dorit.
Constructorul clasei poate avea două forme: *CyclicBarrier (int num), Cyclic Barrier (int num, Runnable barierAction)* în care num reprezintă numărul firelor de execuție, iar la atingerea acestui prag se execută metoda *run()* din obiectul *barrierAction* transmis.

Cele mai importante dintre metodele acestei clase sunt:
- *int await()* – suspendă thread-ul până la atingerea pragului corespunzător obiectului CyclicBarrier
- *int getNumberWaiting()* – returnează numărul de threaduri suspendate la barieră
- *int getParties()* – returnează valoarea parametrului num
- *void reset()* – reinițializează bariera

Un exemplu de utilizare al clasei CyclicBarrier :

```
 public class TechLead extends Thread{
       CyclicBarrier cyclicBarrier;

       public TechLead(CyclicBarrier cyclicBarrier, String name)
       {super(name);
              this.cyclicBarrier= cyclicBarrier; }
       public void run()
       {
              try {
       Thread.sleep(3000);
       System.out.println(Thread.currentThread().getName()+    "    recruited
developer");
       System.out.println(Thread.currentThread().getName()+   "   waiting   to
complete ...");
              cyclicBarrier.await();
              System.out.println("All     finished     recruting,     "     +
             Thread.currentThread().getName()   +
              " gives offer letter to candidate");
              } catch (Exception e) {
                     e.printStackTrace();} }}


public class HRManager {
public static void main(String[] args) {
    CyclicBarrier cyclicBarrier = new CyclicBarrier(3);
    TechLead techLead1 = new TechLead(cyclicBarrier,"John TL");
    TechLead techLead2 = new TechLead(cyclicBarrier,"Doe TL");
    TechLead techLead3 = new TechLead(cyclicBarrier,"Mark TL");
    techLead1.start();
    techLead2.start();
    techLead3.start();
    System.out.println("No work");}}
```

f)Mecanismul Fork/Join

Framework-ul *Fork/Join,* apărut odată cu versiunea Java 1.7 fiind o implementare a interfeței *ExecutorService* care oferă posibilitatea folosirii la maxim a *multiplelor procesoare disponibile*, pentru îmbunătățirea performanței aplicației create. Se utilizează, în general, pentru task-uri recursive.

Ca şi celelalte instanțe ale interfeței *ExecutorService* din pachetul *java.util.concurrent*,ambele *Executor și Fork/join* creează un set de thread-uri pentru distribuirea task-urilor. Noutatea adusă de acest framework este tehnica *work-stealing.* Astfel, thread-urile care execută taskuri extra aplicație pot ”fura” task-uri de la alte thread-uri încă ocupate, pentru a ajuta la terminarea mai rapidă. Atunci când un thread rămâne fără task-uri în coadă, preia task-uri din coada altui thread. Pentru a evita apariția blocajelor, acesta va lua task-uri de la capătul cozii şi va lăsa thread-ului gazdă un semnal prin care îl anunță că i-a preluat task-uri. Nucleul acestui framework este reprezentat de clasa *ForkJoinPool* în care apare implementarea algoritmului work-stealing şi care poate executa procese ForkJoinTask. Această metodă este recomandată în cazul unor operații complexe.

O aplicabilitate imediată a tehnicii Fork/join este sortarea seturilor foarte mari de date, de exemplu sortarea paralelă a vectorilor. Odată cu versiunea Java 1.8 s-au introdus metode noi în cadrul clasei *java.util.Arrays* care utilizează algoritmul fork/join. Principala metodă introdusă este *parallelSort* care poate fi apelată cu diferite argumente. Algoritmul de sortare este un algoritm de tip merge-sort paralel care împarte vectorul în subvectori care sunt la rândul lor sortați şi apoi se compune rezultatul. Atunci când dimensiunea unui subvector atinge un anumit prag, se folosește metoda *Arrays.sort* pentru acel set de date.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

public class ParallelSort {

public static void main(String[] args) {
 List<Integer> list = new ArrayList();
 Random r = new Random();
     for(int x=0; x < 30000000; x++)
     { list.add(r.nextInt(10000));}

     Integer[] array1 = new Integer[list.size()];
     Integer[] array2= new Integer[list.size()];
     array1 = list.toArray(array1);
     array2= list.toArray(array2);

     long start= System.currentTimeMillis();
     Arrays.sort(array1);
     long end = System.currentTimeMillis();
     System.out.println("Sort Time: + " + (end - start));
     start = System.currentTimeMillis();
     Arrays.parallelSort(array2);
     end = System.currentTimeMillis();
     System.out.println("Parallel Sort Time: "+ (end - start))}}
```

h)Streamuri

Prin Stream se înțelege o secvență continuă de elemente care suportă operații agregate secvențiale şi paralele, din acest motiv, odată cu Java 8 a fost necesară introducerea unei noțiuni noi - expresiile lambda. O expresie lambda constă:

- dintr-o listă de parametri formali, separaţi prin virgulă şi cuprinşi eventual între paranteze rotunde,
- săgeata direcţională **->**
- un corp ce constă dintr-o expresie sau un bloc de instrucţiuni.

Un exemplu de utilizare al expresiilor lambda:

```
System.out.println("=== Sorted Asc ===");
Collections.sort(personList, (Person p1, Person p2)->
p1.getSurname.compareTo(p2.getSurname()));
for(Person p: personList)
   {p.printName();}

System.out.println("=== Sorted Desc ===");
Collections.sort(personList, (Person p1, Person p2)->
p2.getSurname.compareTo(p1.getSurname()));
for(Person p: personList)
   {p.printName();}
```

Atunci când stream-urile sunt executate în paralel, Java partiționează execuția pe mai multe substreamuri. Operațiile agregate iterează prin aceste substream-uri şi le procesează în paralel, apoi combină rezultatele. Pentru crearea unui ParallelStream se apelează operația *Collection.parallelStream.* Spre exemplu, în următoarea secțiune se calculează, în paralel, media de vârstă a tuturor bărbaților:

```
Double average= median
.parallelStream()
.filter(p->p.getGender = Person.SEX.MALE)
.mapToInt(Person::getAge)
.average()
.getAsDouble();
```

Tot parallelStream se poate utiliza şi la sortarea vectorilor. Pentru aceasta se apelează la metoda *forEachOrdered* ,ca în următorul exemplu :

```
System.out.println("With forEachOrdered:");
    listOfIntegers.parallelStream().forEachOrdered->System.out.println(e + " ");
    System.out.println("");
```

2.2.6. Elemente de analiză de performanță

Programarea paralelă / concurentă este extrem de utilă la execuția cât mai rapidă și eficientă a unor task-uri complexe. Dacă se face o comparație între rularea serială a unei soluții și rularea folosind threaduri, odată cu creșterea complexității se observă eficiența acestora.

Uneori pentru a face alegerea corectă între cele două variante e nevoie de cunoașterea mai detaliată a unor aspecte legate de performanță, date referitoare la *timp de excuție, memorie utilizată*, nuclee etc. Pentru o analiză corectă și detaliată se pot folosi diferite tool-uri existente care oferă toate informațiile necesare. Pentru Java s-au dezvoltat numeroase astfel de tool-uri, care pot fi fie integrate în mediul de lucru (ex NetBeans Profiler, Eclipse Memory Analizer), fie utilizate individual. O listă de astfel de tool-uri și detalii despre cum pot fi descărcate și utilizate se găsește la adresa https://blog.idrsolutions.com/2014/06/java-performance-tuning-tools/.

Pentru afișarea detaliilor despre memoria folosită și despre numărul de thread-uri folosite vom folosi un tool asemănător cu cel de analiză a concurenței din Visual Studio numit *Netbeans Profile.* Acesta este inclus Netbeans însă necesită anumite configurări, astfel :din meniul *Profile* se alege *Advanced Commands->Run Profiler Calibration* și se alege JDK-ul folosit pentru calibrare.

Pentru a rula analizatorul pentru un anumit proiect se setează pentru proiectul respectiv argumentele care ar trebui sa le primească metoda main() prin vectorul de stringuri ‘args’. Pentru a configura Profiler-ul pentru un anumit proiect se selectează proiectul dorit, apoi din meniul - Profile se alege *Profile Project* și se observă statisticile de analiză pentru proiectul dorit.

2.3. Exemple

2.3.1.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului Java Concurency pentru următoarele mecanisme specifice : threadpool, respectiv semafor, exemple pe care le puteți descărca de la adresa *https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/02_Java_concurent.*

2.4. Întrebări teoretice

2.4.1. Explicați și ilustrați aplicabilitatea a două structuri de date thread-safe identificate în Java Concurrency.

2.4.2.Explicați cele două mecanisme de gestionare a threadurilor ThreadFactory și Threadpoolexecutor.

2.4.3. Explicați și exemplificați conceptul de decuplare a lansării taskurilor de politica lor de execuție.

2.4.4. Ce este un Reentrant lock și ce avantaje oferă ?

2.5. Probleme propuse

2.5.1. Să se implementeze mecanismul de sincronizare Producător-Consumator și să se testeze performanța soluției variind tipul de coadă ales și dimensiunea sa, având drept punct de pornire exemplul prezentat în lucrare.

2.5.2. Să se implementeze un server concurent ce deserveşte un număr nelimitat de clienţi folosind FW Executor cu parametrizare adecvată. Se va utiliza analizorul de performanţă Netbeans Profiler şi se va urmări modificarea timpului de execuţie ca urmare a modificării parametrilor din frameworkul Executor.

2.5.3. Să se studieze,testeze şi evalueze folosind analizorul de performanţă Netbeans Profiler implementările paralele optimizate ale algoritmilor de sortare QuickSort şi MergeSort, identificând cele mai adecvate mecanisme şi structuri de date din frameworkul JavaConcurency cu ajutorul cărora au fost implementaţi (sursa algoritmilor :http://heim.ifi.uio.no/~arnem/sorting/). Analiza se va efectua luând drept criterii timpul de execuţie şi scalabilitatea.

2.6. Miniproiect

Să se implementeze folosind facilităţile moderne de concurenţă din Java **modelul boss-worker** descris în continuare.

Modelul boss-worker este cel mai potrivit pentru aplicatiile de tipul producător - consumator.
In acest model, un thread este desemnat ca boss, toate celelalte fiind desemnate ca workers. Threadul boss obţine sau produce sarcini şi le plasează într-o coadă. Threadurile worker preiau sarcinile din coadă şi le procesează. Exemple de aplicaţii în care acest model este recomandat: aplicaţii care primesc input din surse externe(ex. servere, interfeţe cu utilizatorul), algoritmi de căutare, algoritmi de tipul divide et impera, procesarea buclelor în paralel. Deşi în general există un singur thread boss, nimic nu impiedică o aplicaţie sa creeze mai multe threaduri boss. De asemenea, oricare dintre threadurile worker poate acţiona ca un thread boss, plasând sarcini în coada de sarcini ,de exemplu ca parte din munca de procesare pe care trebuie să o execute workerul respectiv, el poate să creeze sarcini suplimentare.
Pentru implementarea modelului boss-worker se utilizeaza în general un **"thread-pool".** Astfel thread-ul (thread-urile) boss produce sarcini într-o coadă pe care apoi thread-urile worker le procesează. Un threadpool adresează două probleme:

1. îmbunatăţirea performanţelor în cazul în care un număr mare de taskuri sunt executate prin reutilizarea threadurilor worker;
2. limitarea şi managementul resurselor utilizate pentru execuţia unei colecţii de sarcini (taskuri).

Modelul boss-worker în abordarea clasică, anterior JDK 1.5 se implementează utilizînd: două variabile condiţionale ,un mutex ,o coadă ,un număr de contoare .Mutexul protejează coada, variabilele condiţionale şi contoarele. O variabilă conditională este folosită de threadurile worker în aşteptare atunci când coada este goală. Cealaltă variabilă condiţională este utilizată de threadul boss atunci când coada este plină. Un minimum de 3 contoare este necesar pentru a cunoaşte în permanenţă: C1-numărul de sarcini existente în coadă, C2-numărul de threaduri worker in asteptare C3-numărul de threaduri boss în asteptare

Pentru a plasa noi sarcini în coadă, threadul boss trebuie întâi să preia controlul asupra mutexului. In cazul în care coada nu este plină, threadul boss adaugă sarcini în coadă şi semnaleaza acest lucru eventualelor threaduri worker care aşteptau. In cazul în care coada este plină, threadul boss aşteaptă la o variabilă condiţională până când se face loc în coadă pentru noi sarcini. Odată ce sarcinile au fost depuse în coadă, mutexul este eliberat şi threadul boss se reîntoarce la munca sa (adica la crearea sau obţinerea de noi sarcini). Pentru a prelua sarcini din coadă, un thread worker trebuie întâi să preia controlul asupra mutexului. Dacă există sarcini disponibile, atunci una dintre ele este preluată şi acest lucru este semnalat eventualelor threaduri boss care aşteptau. Dacă nu exista sarcini in coada (coada este vida), threadul worker asteapta la o variabilă condiţională până când apar sarcini în coadă. Odată ce sarcina a fost preluată din coada, mutexul este eliberat şi threadul worker începe să proceseze sarcina sa. Sincronizarea este necesară numai atunci când au loc operaţii cu coada, adică atunci când se pun sau se iau sarcini din coadă. Acesta este singurul loc (adică singura structură de date partajată) unde datele pot fi accesate de mai multe threaduri. In timp ce threadul boss creează sarcini, nici unul dintre threadurile worker nu poate accesa aceasta structură sau datele asociate ei. Odată ce un thread worker şi-a preluat sarcina din coadă, nici unul din celelalte threaduri (nici worker, nici boss) nu mai poate accesa acea sarcină sau datele asociate acesteia.

Dimensiunea cozii:poate fi statică sau dinamică. Atunci când coada are o lungime statică sau maximă predefinita, sunt necesare două variabile condiţionale (una pentru threadurile worker, cealaltă pentru threadurile boss). Există situaţii în care lungimea cozii poate fi infinită. In aceste cazuri, este suficientă o singură variabilă

condițională și anume cea pentru threadurile worker în așteptare, deoarece threadurile boss nu se pot bloca, din moment ce coada nu se va umple niciodată. Totuși, cozile statice au avantajul că threadul boss nu supraîncarcă threadurile worker cu mai multă muncă decât acestea pot procesa. O oarecare balansare este asigurată folosind cozi cu lungime statică. Dezavantajul acestei situatii este cel menționat și anume ca threadul boss se poate bloca, fapt ce poate avea diferite impacte asupra utilizatorului sau timpului de răspuns al aplicației.

In modelul boss-worker, fiecare dintre threadurile worker se află într-o buclă continuă în care obțin și proceseaza sarcini. La un moment dat, toate sarcinile se vor fi terminat și threadurile worker ar trebui să se termine și ele. Acestea nu se pot termina pur și simplu când coada este goală, ci trebuie anunțate în mod explicit că trebuie să se termine.

Exista două tehnici de realizare a acestui lucru:

1. sa se creeze o sarcină specială, de "exit", pentru fiecare dintre threadurile worker, și fiecare dintre aceste sarcini să fie plasată în coadă. Când un thread worker primește o sarcină de ieșire (exit task), el se va termina.

   Observație: acest lucru este desigur valabil dacă ordinea de prelucrare a sarcinilor este FIFO, deoarece sarcinile de ieșire trebuie să fie ultimele sarcini depuse de threadul boss în coadă.

2. să se asocieze cozii un flag "exit". Atunci când threadul boss dorește ca threadurile worker să se termine, el seteaza acest flag la valoarea EXIT.

Când threadurile worker încearcă să extragă o sarcină din coadă,în cazul în care coada este vidă și flagul de exit este setat, atunci ele ies din bucla infinită și se termină.

Observație: coada trebuie sa fie vidă. Daca threadurile worker s-ar termina oricând flagul este setat, atunci ele s-ar putea termina înainte ca toate sarcinile să fi fost prelucrate. Odată ce toate threadurile worker au fost instruite să se termine, threadul boss așteaptă terminarea tuturor threadurilor worker (join).

Pentru aplicațiile care folosesc acest model pentru implementarea concurenței, echilibrarea încărcării nu reprezintă o problemă. Aceste aplicații pot folosi modelul boss-worker pentru a împarți sarcinile astfel încât atunci când un thread se blochează, un altul poate continua în locul lui. Acest caz este cel mai des întâlnit atunci cînd modelul boss-worker este folosit pentru servere și interfețe cu utilizatorul, unde cantitatea de sarcini este controlată de evenimente externe.Tthreadul boss acceptă taskuri și le pasează threadurilor worker, iar planificarea balansării nu este o problemă pentru aceste aplicații.

Un alt tip de echilibrare este totusi important în modelul boss-worker: echilibrarea cozii. O aplicație eficienta va implementa cozi care nu produc nici blocarea threadurilor boss, nici blocarea threadurilor worker (mai explicit, coada nu va fi niciodată nici goală, nici plină). Pentru obținerea acestei situații ideale, se poate experimenta în sensul ajustării *numărului de sarcini, a numărului de threaduri worker, a dimensiunii cozii sau a dimensiunii sarcinilor* (acolo unde acest lucru este posibil; există cazuri în care dimensiunea sarcinii este fixată); de exemplu, dacă threadul boss se blochează datorită umplerii cozii, se poate mări numărul de threaduri worker sau se poate mări dimensiunea cozii; dacă threadurile worker se blochează datorită faptului că în coadă nu există sarcini, se poate mări numărul de threaduri boss sau micșora numărul de threaduri worker.

In cazul aplicațiilor care sunt controlate de evenimente externe (de exemplu, input de la utilizator sau de pe rețea), o balansare a cozii nu va fi niciodată posibilă. Aceste aplicații ar trebui să aibă in vedere crearea unui număr dinamic de threaduri boss și threaduri worker. Când volumul de muncă existent este redus, se vor folosi un singur thread boss și câteva threaduri worker; când volumul de muncă sporește, se mărește și numărul de threaduri worker și eventual și cel de threaduri boss.

Pachetul **java.util.concurrent din JDK 1.7** oferă (printre multe alte facilități) metode de implementare a modelului discutat.

Astfel, clasa java.util.concurrent.ThreadPoolExecutor poate fi utilizată pentru crearea unui thread-pool. Următorii parametrii pot fi specificați la instanțierea unui astfel de obiect:

- **corePoolSize** - numărul minim de fire de executie "worker" ce sunt menținute în pool.
- **maximumPoolSize** - numărul maxim de fire de execuție ce pot fi create.
- **keepAliveTime** - când numărul de threaduri din pool este mai mare decat corePoolSize, acest parametru specifică timpul maxim în care firele de execuție în exces vor astepta sarcini înainte de a se termina
- **unit** - unitatea de timp (TimeUnit) pentru parametrul keepAliveTime

- **workQueue** - coada utilizată pentru salvarea sarcinilor ce urmează să fie executate. O sarcină este un obiect ce implementează interfata Runnable și poate fi introdusă în coadă utilizand metoda execute.

Coada trebuie să implementeze interfața BockingQueue<E> Implementări existente pentru o coadă pe care puteți să le utilizati:

ArrayBlockingQueue -coadă de dimensiune fixă ce utilizeaza un vector pentru salvarea sarcinilor
LinkedBlockingQueue -coadă de dimensiune nelimitată implementata printr-o listă simplu inlanțuită (FIFO)
PriorityBlockingQueue -coadă de dimensiune nelimitată în care se poate defini ordinea de inserare a elementelor
SynchronousQueue -coadă fara dimensiune, adaugarea respectiv extragerea elementelor din coada presupunand blocare.

- **threadFactory** - mecanismul de instanțiere a firelor de executie worker
- **handler** - handler utilizat pentru a primi notificari in cazul in care un task nu se poate executa.

Astfel , o sarcină va fi rejectată dacă e îndeplinită una din condițiile :

- mecanismul de introducere de sarcini a fost oprit (shutdown)
- pool-ul de threaduri este saturat (s-a ajuns la limita de maximumPoolSize threaduri și s-a atins limita maximă a cozii de sarcini (dacă se utilizează o coadă de dimensiune maximă fixată))

Schelet de implementare . Instanțierea unui thread-pool ce utilizează o coadă de sarcini de dimensiune infinită, 4 threaduri de bază și stabilește o limită maximă de 20 threaduri ce pot rula.

```
ExecutorService myPool =     new ThreadPoolExecutor(4, 20,
1L,TimeUnit.MILLISECONDS newLinkedBlockingQueue<Runnable>());,
```

Utilizare:

```
while (exista_sarcini){
     [producere sarcini]
     Runnable sarcina = new PoolTask(sarcina);
myPool.execute(sarcina) }
..
   [finish]
   myPool.shutdown();
   try {
   allJoined = myPool.awaitTermination(60L,TimeUnit.SECONDS);
if(!allJoined)
                myPool.shutdownNow()
        } catch (Exception e) {//nothing to do}

Sarcina (task):
class PoolTask implements Runnable{
    Sarcina sarcina;
   PoolTask(Sarcina s){
this.sarcina=s; }
   void run(){
   //procesare sarcina }}
```

2.7. Referințe bibliografice

1. API :https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html
2. Tutorial Oracle : http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
3. Tutorial concurență https://howtodoinjava.com/java/multi-threading
4. Exemple aplicații cu implementare concurentă : http://www.cdac.in/index.aspx?id=ev_hpc_hypack_java_concurrent_programs
5. Martin Mois- Java Concurrency essentials, JDC books,2015 disponibilă la http://enos.itcollege.ee/~jpoial/allalaadimised/reading/Java-Concurrency-Essentials.pdf

3.OpenMP- model de programare paralelă a sistemelor cu memorie partajată

OpenMP este cel mai răspândit model pentru programarea paralelă a maşinilor cu memorie partajată, oferind paralelism de tip fork-join explicit în care un thread master creează şi coordonează multiple threaduri, model parametrizabil prin intermediul unor construcţii de tip directive şi clauzele asociate acestora , respectiv utilizând variabile de mediu şi funcţii de bibliotecă pentru crearea de taskuri paralele/concurente cu execuţie controlată prin construcţiile de sincronizare.

3.1. Obiective

- Studiul modelului de programare paralelă OpenMP
- Implementarea paralelă a unor algoritmi în modelul cu memorie partajată

3.2. Concepte

Modelul OpenMP a fost lansat în 1997 (gestionat de OpenMP ARBwww.openmp.org) şi a evoluat continuu ajungând în prezent la versiunea 5.0.(2018) , fiind utilizat pentru dezvoltarea aplicaţiilor paralele pe platforme diverse de la sisteme desktop până la supercomputere. În cadrul acestui model, threaduri cooperante rezolvă sarcini (taskuri) prin execuţie simultană pe procesoare sau core-uri multiple.
Modelul de programare OpenMP se bazeaza pe:

- *memorie partajata şi paralelism bazat pe threaduri* : un proces cu memorie partajată poate consta în fire de execuţie multiple. OpenMP se bazează pe existenţa firelor multiple în paradigma programării cu partajarea memoriei.
- *paralelism explicit*: OpenMP este un model de programre explicit (nu automat), care oferă programatorului deplinul control asupra paralelizării.
- *modelul fork-join* : OpenMP uzează de un model al execuţiei paralele alcătuit din ramificaţii şi joncţiuni vezi (vezi Figura 3.1.)

Directivele compilator sunt utilizate ca extensii pentru limbajele secvenţiale C++/Fortran oferind construcţii pentru creare de taskuri, partajare sarcini , sincronizarea threadurilor .

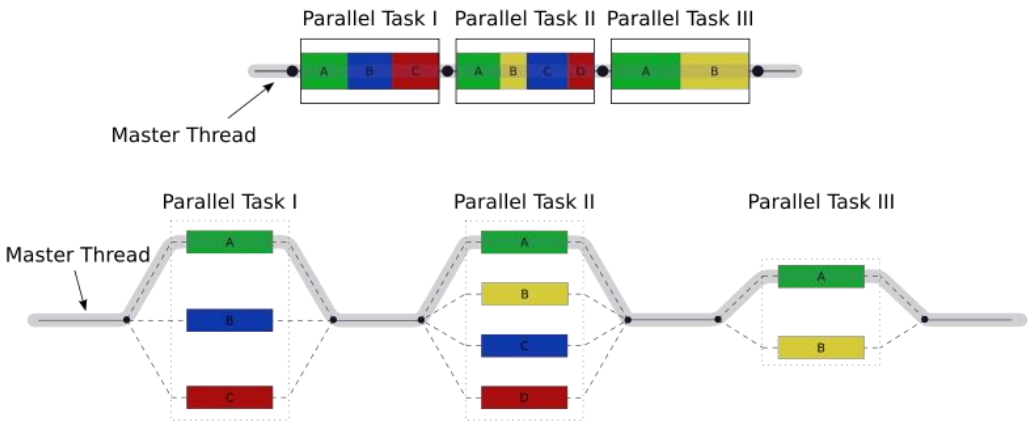


Figura 3.1. Modelul fork-join OpenMP [1]

Setul de directive OpenMP permite specificarea acţiunilor compilator necesare execuţiei codului program în paralel fără însă a verifica *dependenţele, conflictele, blocajele* posibile, programatorul fiind astfel responsabil în totalitate de proiectarea corectă a aplicaţiei. Modelul nu este aplicabil sistemelor cu memorie distribuită, nu este implementat în mod identic de către toţi producătorii (de ex. Intel/Microsoft) şi nu este garantat că ar asigura cea mai eficientă utilizare a memoriei partajate deoarece nu există pentru moment constructori de localizare a datelor , dar este de aşteptat a deveni în viitor un standard ANSI.

3.2.1. Modelul de programare şi modelul de memorie

Standardul OpenMP oferă o interfaţă (API) de *programare independentă de platformă* (există implementări Linux/Windows), integrând o serie de construcţii pentru managementul proceselor, distribuţia taskurilor, concurenţă şi sincronizare a threadurilor şi gestionarea datelor. Modelul de programare oferă deasemenea construcţii pentru controlul partajării şi consistenţei memoriei utilizate de aplicaţie şi suport pentru paralelism imbricat.

Regiunea paralelă este construcţia ce defineşte o secţiune de program sub forma unui *bloc structurat* (marcat prin intermediul acoladelor*), executat în paralel de procesoarele sistemului.* Programul îşi începe execuţia cu un singur thread, iar în momentul întâlnirii primei regiuni paralele, thread-ul master, având id-ul 0 , creează o echipă de thread-uri (după modelul fork / join prezentat în figura 3.1.) şi se include pe sine în echipă, threaduri ce vor executa regiunea paralelă prin planificarea (alocarea) iteraţiilor ciclurilor paralele acestor threaduri. Această operaţie de tip fork presupune că la finele regiunii paralele există o barieră *implicită* pentru sincronizare şi doar threadul master continuă execuţia ulterior acesteia. *Numărul de threaduri* poate fi specificat în directivă, stabilit folosind variabile de mediu sau în mod dinamic la runtime prin funcţii OpenMP. Dacă un thread modifică un obiect partajat, acest fapt va afecta nu doar propriul context de execuţie ci şi pe cele ale celorlalte threaduri ale programului care au acces şi pot utiliza respectivul obiect. Asignarea specifică de taskuri diferitelor threaduri este deasemenea posibilă prin intermediul unor directive specifice ce vor fi prezentate în continuare.

Modelul de memorie distinge între *memorie partajată şi memorie privată*, toate threadurile având acces la memoria partajată (modelul implicit), iar pentru evitarea blocajelor sau a conflictelor sunt prevăzute mecanisme de sincronizare. Complementar memoriei partajate, un thread poate utiliza variabile private în memoria privată proprie, variabile ce nu pot fi accesate de alte threaduri.

Extensiile de limbaj oferite de standardul OpenMP sunt structurate în următoarele categorii : control paralel, partajare sarcini, mediu de date, sincronizare funcţii runtime şi variabile de mediu (vezi Figura 3.2.) şi vor fi abordate în continuare.

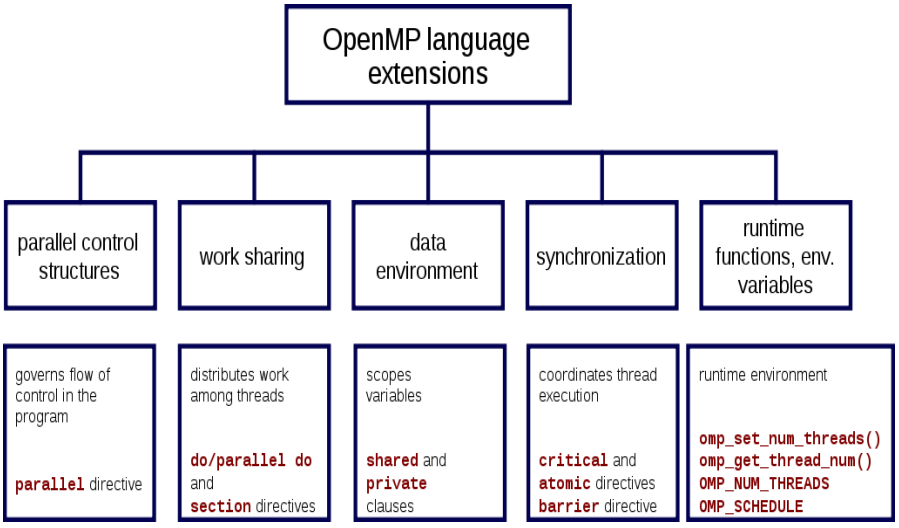


Figura 3.2. Tipuri de construcţii OpenMP [1]

Directive. O directivă este o linie specială de cod sursă având semantică utilă doar anumitor compilatoare ce integrează suport pentru expandarea şi execuția sa. Directivele sunt case - sensitive, iar ordinea execuţiei clauzelor directivei este nesemnificativă.

Orice directivă este de forma: **#pragma omp.** Sintaxa directivelor este:

#pragma omp *[clauză[[,]clauză]. . .]linie nouă*

Clauze. Specificarea de informaţii adiţionale în diverse directive, inclusiv directiva de creare a unei regiuni paralele este realizată prin *clauze* ce pot fi specificate sub forma unei liste. Lista clauzelor permite specificarea paralelizării condiţionale, a numărului de threaduri de execuţie şi a modului de gestionare a datelor, cele mai importante categorii de clauze fiind:

- Specificarea datelor : acestea pot fi date partajate şi private. În regiunile paralele, variabilele pot fi *partajate sau private.* Toate thread-urile văd aceeaşi copie a variabilelor partajate, pot citi sau scrie variabilele partajate, însă fiecare thread are propria copie a variabilelor private, invizibile pentru celelalte thread-uri, astfel o variabilă privată poate fi citită sau scrisă numai de thread-ul căruia îi aparţine.
- Specificarea ciclurilor paralele. Ciclurile sunt principala sursă de paralelism, dacă iteraţiile unui ciclu sunt *independente* (pot fi executate în orice ordine), atunci se pot partaja iteraţiile între thread-uri diferite.
- Specificare unor operații pe date ,de exemplu reducţii. Reducţia este operaţia prin care se generează o singură valoare din operaţii asociative. Permiţând unui singur thread la un moment dat să actualizeze o variabilă partajată. Aceasta presupune eliminarea totală a paralelismului, dar este posibil ca fiecare thread să acumuleze în propria copie privată, iar apoi aceste copii sunt reduse pentru a furniza un rezultat final.

Compilare condiţională. Directivele respectă convenţiile din standardele C/C++ pentru directivele de compilare. Sensibilă la upper/lower case. Pentru o directivă, numai un nume de directivă poate fi specificat. Fiecare directivă se aplică la cel mult o declaraţie următoare, care poate fi un bloc structurat. Liniile-directivă lungi pot fi continuate pe linii următoare prin combinarea caracterelor newline (linie-nouă) cu un caracter "\" la finalul liniei-directivă.

Domeniile de aplicabilitate a directivelor sunt :

- *domeniu static (lexical):* Codul inclus textual între începutul si finalul unui bloc structurat care urmează directiva. Domeniul static al unei directive nu acoperă rutine multiple sau fisiere de cod.
- *directive orfane:* Despre o directivă OpenMP care apare independent de o altă directivă care o include se spune că este o directivă orfană. Ea există în afara domeniului static (lexical) al altei directive. Acoperă rutine si posibile fisiere de cod.
- *domeniu dinamic:* Domeniul dinamic al unei directive include atât domeniul ei static (lexical) cât si domeniile orfanelor ei

Variabile de mediu. Variabilele de mediu permit controlul execuţiei paralele. Cele mai uzuale sunt:

OMP_NUM_THREADS -setează numărul de threaduri posibil a fi utilizate în timpul execuţiei programului.

OMP_SCHEDULE - setează tipul planificării run-time a execuției threadurilor (cu alocare de un anumit tip a taskurilor a threadurilor)

OMP_DYNAMIC - activează-dezactivează ajustarea dinamică a numărului de threaduri care pot fi utilizate pentru executarea unor zone diferite în paralel

OMP_NESTED - activează-dezactivează paralelismul imbricat (utilizarea imbricată a directivelor de tip parallel for)

Variabile partajate şi private. În interiorul unei regiuni paralele variabilele pot fi ***partajate*** *-toate thread-urile văd aceeaşi copie (modelul implicit),* sau ***private*** *- fiecare thread are propria copie asupra căreia operează* şi al cărei conţinut poat deveni accesibil ulterior celorlalte threaduri.
Clauzele SHARED, PRIVATE, DEFAULT pot fi utilizate pentru specificarea tipului variabilelor, conform sintaxei: *shared(list), private(list), default(shared|none).*

Câteva observaţii se impun referitor la declararea tipurilor variabilelor:

- majoritatea variabilelor sunt partajate (este modelul implicit)
- indicii ciclurilor sunt privaţi
- variabilele temporare ale ciclurilor sunt private

3.2.2. Regiuni paralele

Codul sursă din interiorul unei regiuni paralele este executat de toate thread-urile. Sintaxa directivei de creare a unei regiuni paralele este:

#pragma omp parallel *[clauză[[,]clauză]. . .]linie nouă*
bloc_ structurat

iar clauza poate lua una din valorile

if(*expresie _scalară***)**
private(listă _var**)**
firstprivate(listă _var**)**
default(shared | none)
shared(listă _var**)**
copyin(listă _var**)**
reduction(*operato***r:** listă _var**)**
num_threads(*expresie int.* **)**

Numărul de threaduri dintr-o regiune paralel este determinat de factorii următori, în ordinea prezentată: (1)se utilizează functia de bibliotecă omp_set_num_threads(), (2)se setează variabila de mediu OMP_NUM_THREADS, (3)implementarea default.

Există o serie de funcţii complementare cu ajutorul cărora poate fi controlată execuţia în cadrul regiunilor paralele prin stabilirea/determinarea numărului de threaduri active la un moment dat, astfel:

int omp_get_num_threads(void) - determină numărul de threaduri, funcţia returnează 1 dacă este apelată în exteriorul unei regiuni paralele

int omp_get_thread_num(void) - determină numărul threadului curent ,ia valori între 0 şi OMP_GET_NUM_THREADS() – 1

int omp_set_num_threads(num_threads) - setarea numărului de threaduri utilizabile în regiunile paralele următoare

int omp_get_max_threads(void) - funcţie ce returnează limita superioară a numărului de threaduri ce poate fi creat de directiva PARALLEL

int omp_get_num_procs(void) - funcţie ce returnează numărul maxim de procesoare ce pot fi utilizate pentru execuţia programului

Fire dinamice. Implicit, un program cu regiuni paralele multiple utilizează acelasi număr de fire pentru a executa fiecare dintre regiuni. Această comportare poate fi modificată pentru a permite la momentul executiei modificarea dinamică a firelor create pentru o anumită sectiune paralelă. Cele două metode disponibile pentru a permite fire dinamice sunt: fie utilizarea functiei de bibliotecă omp_set_dynamic(), respectiv setarea variabilei de mediu OMP_DYNAMIC.

Iniţializarea variabilelor private. Atunci când o variabilă este declarată privată, threadul primeşte adresa unde va stoca valorile variabilei pe durata regiunii paralele. La incheierea regiunii paralele, memoria este dealocată şi aceste variabile nu mai există. Dacă se doreşte reţinerea valorii acestor variabile anterior/ulterior regiunii paralele se vor folosi clauzele FIRSTPRIVATE şi

LASTPRIVATE. Variabilele private sunt neiniţializate la începutul regiunii paralele. Dacă se doreşte iniţializarea lor, se poate folosi clauza FIRSTPRIVATE: firstprivate(*list*).

Clauza de reducţie. O reducţie produce o singură valoare din operaţii asociative precum: adunarea, înmulţirea, max, min, ŞI (logic), SAU (logic). Operaţia este echivalentă următorului set de etape: fiecare thread reduce într-o copie privată, iar apoi toate aceste copii se reduc pentru a furniza rezultatul final, clauza utilizată este REDUCTION. Este permisă utilizarea şirurilor ca variabile de reducţie.
Sintaxa clauzei este*: reduction(op:list)*

Clauza IF. Directiva de regiune paralelă poate fi condiţională, specificare utilă situaţiilor în care nu există suficientă sarcină de procesare pentru a utiliza efectiv paralelismul.
Sintaxa este: *if (scalar expression)*

3.2.3. Directive de partajare a lucrului

Directivele de partajare a lucrului (FOR, SECTIONS, SINGLE) distribuie execuţia între membrii echipei de threaduri. Aceste directive sunt plasate în *interiorul unei regiuni paralele* şi indică modul în care este divizat lucrul. Directivele nu lansează noi threaduri, iar la începutul unei asemenea construcţii nu se consideră în mod implicit plasarea unei bariere, secvenţa de construcţii de partajare a lucrului şi directivele de tip barieră trebuie să fie similare pentru grupul de threaduri.

Directiva FOR. Deoarece ciclurile sunt cea mai comună sursă de paralelism în majoritatea programelor, ele permit *divizarea iteraţiilor ciclului între thread-uri.* Ciclurile identifică o construcţie de partajare a sarcinilor iterativă ce permite specificarea execuţiei în paralel a iteraţiilor asociate ciclului şi distribuite threadurilor deja existente**.**

Sintaxa este:
#pragma omp for *[clauză [,] clauză]. . .] linie _nouă*
for-loop

iar clauzele pot fi:
private(listă _var**)**
firstprivate(listă _var**)**
lastprivate(listă _var**)**
reduction(*operato***r:** listă _var**)**
ordered
schedule(*tip***[,** *chunk_size***])**
nowait

Semnificaţia clauzelor în contextul acestei directive este următoarea: *firstprivate* specifică copii locale ale variabilei private threadului, iar *lastprivate* permite specificarea actualizării variabilei la ultima iteraţie a ciclului for. Directiva plasează restricţii asupra structurii buclei for corespondente, astfel ,aceasta trebuie să respecte o formă canonică: *for (var = a; var op_logic b; incr-exp)*

unde op_logic poate fi: <, <=, >, >=
şi incr-expr este var = var +/- incr sau echivalenţi semantici precum var++, ce nu poate fi modificată în corpul ciclului. Forma canonică permite ca numărul de iteraţii să fie determinat la intrarea ciclului.

Deoarece construcţia este larg utilizată, există o formă scurtă care combină regiunea paralelă cu directivele Do/For astfel rezultă construcţia combinată:

#pragma omp parallel for *[clauze]*
for loop

Fără clauze adiţionale, directiva Do/For va *partiţiona iteraţiile în mod echilibrat* între threaduri, însă implementarea este dependentă de sistem. Un test ce permite verificarea dacă un ciclu este paralel este următorul: se verifică dacă ciclul oferă aceleaşi rezultate dacă este rulat în ordine inversă (atunci este mai mult ca sigur paralel, salturile în afara ciclului nefiind permise).
Variabila de index a ciclului paralel este PRIVATĂ prin definiţie, iar directiva PARALLEL DO/FOR acceptă toate clauzele directivei PARALLEL.

Clauza SCHEDULE oferă o varietate de opţiuni pentru specificarea *divizării iteraţiilor ciclurilor* care vor fi executate de fiecare thread. Sintaxa acesteia este**:**

schedule (*tip[, size]***)**

unde *tip* poate fi: STATIC, DYNAMIC, GUIDED sau RUNTIME, iar *chunksize* este o expresie întreagă pozitivă, ce reflectă modul în care a fost divizat spaţiul de iteraţii.

Cerinţe pentru planificare statică

- dacă *size* nu este specificat, spaţiul de iteraţie este divizat în părţi egale, şi o parte este asignată fiecărui thread (planificare pe block)
- dacă *size* este specificat, spaţiul de iteraţie este divizat în părţi, fiecare parte de un număr egal cu *chunksize* iteraţii, iar părţile sunt asignate în mod ciclic fiecărui thread (planificare tip block-cyclic)

Cerinţe pentru planificare dinamică

- planificarea dinamică divide spaţiul de iteraţie în părţi de dimensiunea *size* şi le asignează thread-urilor după regula primul-venit-primul-servit, astfel dacă un thread termină o parte, îi este asignată următoarea parte din listă.
- dacă *size* nu este specificat, se consideră având valoarea 1.

Cerinţe pentru planificare ghidată (GUIDED)

- este similară celei dinamice, dar secvenţele planificate sunt mari la început şi apoi se micşorează exponenţial.
- dimensiunea următoarei părţi reprezintă (generic) numărul de iteraţii rămase, divizat la numărul thread-urilor.
- câmpul *size* specifică dimensiunea minimă a unei părţi, dacă *size* nu este specificat, se consideră ca fiind 1.

Planificarea *RUNTIME* permite alegerea planificării în momentul rulării, când este determinată de valoarea variabilei de mediu OMP_SCHEDULE.

Observaţii utile în alegerea unei anumite planificări sunt următoarele:

- se utilizează planificarea STATIC pentru cicluri *echilibrate* ca încărcare, utilă pentru cicluri cu încărcare uşoară, dar care poate induce partajare falsă.
- este potrivită planificarea DYNAMIC dacă iteraţiile au *încărcare variabilă* la scară largă, dar distruge localitatea datelor, iar planificarea GUIDED, mai puţin costisitoare decât DYNAMIC oferă posibilitatea unei planificări dirijate.
- în mod uzual se utilizează modul RUNTIME pentru experimentări diverse.

Standardul OpenMP propune şi soluţii de paralelism dinamic implicit**,** astfel este posibil să lăsăm sistemul să decidă câte thread-uri execută fiecare regiune paralelă, pentru ca acesta să realizeze *optimizările* necesare alocării resurselor. Numărul thread-urilor va fi egal sau mai mic decât cel setat de utilizator şi va rămâne fix pe durata fiecărei regiuni paralele.

Paralelismul dinamic se poate seta cu rutina OMP_SET_DYNAMIC, sau cu variabila de mediu OMP_DYNAMIC. Valoarea implicită este dependentă de implementare. Dacă codul depinde

de utilizarea unui anumit număr de thread-uri, atunci este necesară dezactivarea paralelismului dinamic.

Alături de funcţiile ce permit identificarea threadurilor create şi a numărului lor sunt utile funcţiile ce permit *alterarea dinamică* a numărului de threaduri *int omp_get_dynamic()*, respectiv activarea paralelismului imbricat *int omp_get_nested().*

Directiva SECTIONS. Directiva permite partiţionarea spaţiului iteraţiilor între threaduri, în acest mod blocuri separate, independente de cod pot fi executate în paralel (ex. diferite subrutine independente). Prin intermediul acestei dircetive, OpenMP oferă asignarea noniterativă a taskurilor paralele, iar codul sursă este cel ce determină cantitatea de paralelism pusă la dispoziţie. Aceste directive sunt relativ rar utilizate, cu excepţia implementării paralelismului imbricat.

```
#pragma omp sections [clauze]
{
        [ #pragma omp section ]
                bloc structurat
        [ #pragma omp section
                bloc_ structurat
        . . . ]}
```

Directiva *Sections* acceptă clauzele PRIVATE, FIRSTPRIVATE şi LASTPRIVATE, REDUCTION, NOWAIT fiecare secţiune trebuie însă să conţină un bloc structurat. La sfârşitul directivei o barieră implicită realizează sincronizarea ,în absenţa unei clauze NOWAIT. Forma scurtă a acestei directive este:

```
#pragma omp parallel sections [clauze]
        {
        . . . }
```

3.2.4.Directive de sincronizare şi consistenţă a memoriei

Directiva SINGLE. Este necesar ca pentru anumite situaţii o anumită parte de procesare din regiunea paralelă să fie executată de *un singur thread*, fiind utilă pentru calcul de date globale, respectiv operaţii de intrare-ieşire. Primul thread care ajunge la directiva SINGLE va executa blocul, iar celelalte thread-uri vor aştepta până când blocul a fost executat. Directiva acceptă clauzele directivelor PRIVATE şi FIRSTPRIVATE, în mod implicit la sfârşitul directivei se găseşte o barieră. Această directivă este utilă pentru calcule de date globale sau operaţii I-O.

Sintaxa:

```
#pragma omp single [clauze]
bloc structurat
```

iar clauza este una din următoarele:

private(*lista_var*)
firstprivate(*lista_var*)
copyprivate(*lista_var*)
nowait

Directiva MASTER. Directiva Master este o specializare a directivei SINGLE, ea indică faptul că un bloc de cod trebuie să fie executat numai de thread-ul master (thread 0), iar celelalte thread-uri evită acest bloc şi îşi continuă execuţia. Sintaxa:

```
#pragma omp master
bloc structurat
```

Directiva ORDERED. Directiva permite specificarea codului unui ciclu ce trebuie executat *în ordinea în care ar fi fost executat secvenţial*. Deoarece directiva referă o execuţie de tip *in-order* a unui ciclu FOR, aceasta poate apărea numai în interiorul unei directive ParallelFor care are clauza

ORDERED specificată. Directiva introduce un punct de serializare în program, astfel un singur thread poate accesa secvenţa de cod astfel definită, doar atunci când threadurile anterioare au ieşit din ciclu. Sintaxa:

#pragma omp ordered
bloc structurat

Directiva BARRIER. Implementarea unei bariere presupune că nici un thread nu poate trece de barieră până când aceasta nu a fost atinsă şi de către celelalte threaduri. Există o barieră implicită la sfârşitul directivelor DO/FOR, SECTIONS şi SINGLE, astfel fie toate thread-urile, fie nici unul trebuie să atingă bariera, în caz contrar apare o blocare (blocare circulară - DEADLOCK). Sintaxa:

#pragma omp barrier

Clauza NOWAIT. Poate fi folosită pentru a elimina barierele implicite de la sfârşitul directivelor DO/FOR, SECTIONS şi SINGLE (există anumite situaţii în care barierele sunt costisitoare). Directiva indică faptul că threadurile pot trece la instrucţiunea următoare fără a aştepta ca celelalte threaduri să termine execuţia ciclului for. Sintaxa:

#pragma omp for nowait
for loop

Observaţie. Este foarte uşor a se omite utilizarea unei bariere necesare, fiind posibil astfel să se genereze un comportament nedeterminist. Prin folosirea directivei NOWAIT şi explicitarea tuturor barierelor se poate utiliza un stil de programare care să evite un astfel de comportament al aplicaţiei.

Directiva CRITICAL. O secţiune critică este un bloc de cod care poate fi executat de un singur thread la un moment dat şi este folosită pentru a proteja variabilele partajate. Alături de funcţiile de bibliotecă specifice managementului lacătelor, directiva CRITICAL permite denumirea secţiunilor critice, astfel dacă un thread este într-o secţiune critică cu un nume dat, atunci nici un alt thread nu poate pătrunde într-o secţiune critică cu acelaşi nume. Dacă regiunea critică nu este numită secţiunea va primi un nume implicit, numele regiunilor critice este global. Regiunile critice reprezintă zone de serializare în cod ce trebuie reduse pentru creşterea performanţei programului. Sintaxa**:**

#pragma omp critical *[***(** *nume* **)***]*
bloc structurat

Directiva ATOMIC. Directiva este folosită pentru a proteja o singură actualizare a unei variabile partajate, aplicată unui singur bloc. Utilizarea acesteia poate fi mai eficientă decât folosirea directivei CRITICAL, de ex. dacă diferite elemente ale unui şir pot necesita protejare distinctă.

#pragma omp atomic
bloc structurat

Este de preferat utilizarea directivei ATOMIC, deoarece permite cea mai bună optimizare, dacă nu este posibil, se poate folosi directiva CRITICAL, având grijă ca numele date să fie pe cât posibil diferite.

Directiva FLUSH. Oferă un mecanism ce permite implementarea consistenţei memoriei între threaduri diferite, astfel în acest mod întregul set de threaduri posedă o viziune consistentă a anumitor obiecte din memorie necesare aplicaţiei. Dacă obiectele ce necesită sincronizarea sunt desemnate ca variabile, acestea pot fi specificate în lista opţională. Directiva fără o listă de variabile va sincroniza toate obiectele.

#pragma omp flush *[***(***list***)***]*

unde *list* specifică o listă de variabile care trebuiesc golite. Dacă nu este specificată nici o listă, atunci toate variabilele partajate vor fi golite.

O directivă FLUSH este utilizată de o directivă BARRIER la intrarea şi ieşirea dintr-o secţiune CRITICAL sau ORDERED, PARALLEL, PARALLEL FOR şi la sfârşitul directivelor

PARALLEL, DO/FOR, SECTIONS şi SINGLE (mai puţin atunci când este prezentă clauza NOWAIT). Directiva nu este implicită pentru intrarea clauzelor: FOR, MASTER, SECTIONS, SINGLE.

3.2.5. Date în aplicaţii OpenMP

Performanţa programelor este în mod direct influenţată de manipularea datelor de către threaduri. Anumite directive acceptă clauze ce permit utilizatorului controlul atributelor de partajare a variabilelor pentru o regiune, aplicate doar pentru extensia lexicală specifică directivei. Lista clauzelor valide pentru o anumită directivă sunt specifice directivei. Există un set de directive ce permit controlul mediului de date pe durata execuţiei regiunii paralele, cele mai importante fiind THREADPRIVATE ce defineşte domeniul de adresare ca fiind fişier, spaţiu de nume respectiv static şi un set de clauze ce permit controlul atributelor de partajare a variabilelor. Dacă o anumită variabilă este vizibilă la apariţia unei construcţii de partajare a lucrului şi nu este specificată în lista THREADPRIVATE sau într-o clauză de atribute, atunci variabila este partajată.

Directiva THREADPRIVATE. Pentru anumite aplicaţii poate fi convenabil ca fiecare thread să posede *propria copie* a variabilelor cu scop global (variabile file-scope şi namespace-scope în C/C++). În afara regiunilor paralele şi în directivele MASTER, accesele la aceste variabile, referă copia thread-ului master.

Există situaţii în care este necesar ca anumite variabile să fie accesibile prin menţinerea lor persistentă între regiuni paralele, *fără a fi copiate în spaţiul de date al threadului master*. Această clasă de variabile persistente este realizată folosind directiva THREADPRIVATE, ce implică faptul că toate variabilele listei sunt locale fiecărui thread şi sunt iniţializate înainte de a fi accesate într-o regiune paralelă, ele fiind persistente. Utilizarea directivei presupune o serie de restricţii între care, directiva trebuie să aibă scop la nivel de fişier sau de spaţiu de nume, după toate declaraţiile de variabile din *list* şi înainte de orice referinţe la variabilele din *list*.

Sintaxa este:**#pragma omp threadprivate (***list***)**

Clauza PRIVATE. Clauza declară variabilele din listă private pentru fiecare thread, alocând în acest sens un nou obiect persistent ale cărui caracteristici sunt determinate de tipul variabilei. Sintaxa: **private(list)**

Clauza FIRSTPRIVATE. Clauza oferă un superset aferent funcţionalităţii oferite de clauza PRIVATE. Această directivă este utilizată atunci când este necesară valoarea unei variabile private la intrarea într-un ciclu Sintaxa: **firstprivate(list)**

Clauza LASTPRIVATE. Această directivă este utilizată atunci când este necesară valoarea unei variabile private la ieşirea dintr-un ciclu. Sintaxa:**lastprivate(list)**

Clauza SHARED. Clauza defineşte variabile ce apar în listă ca fiind partajate de grupul de threaduri ce pot accesa simultan aceeaşi zonă de stocare a acestora. Sintaxa: **shared(list)**

Clauza REDUCTION. O reducţie produce o singură valoare din operaţii asociative precum: adunarea, înmulţirea, max, min, ŞI (logic), SAU (logic). Operaţia este echivalentă următorului set de etape: fiecare thread reduce într-o copie privată, iar apoi toate aceste copii se reduc pentru a furniza rezultatul final. Sintaxa **reduction(*op*:*list*)**

Clauza COPYIN. Această clauză permite ca valorile datelor private ale thread-ului master să fie copiate la toate celelalte thread-uri la începutul regiunii paralele. Sintaxa: **copyin(***list***)**

Clauza COPYPRIVATE. Clauza oferă un mecanism de utilizare a unei variabile private pentru a transmite în mod broadcast o valoare de la un thread către celelalte threaduri din

grup. Clauza poate apărea doar corelat cu directiva SINGLE. Efectul clauzei asupra listei de variabile apare după execuţia blocului structurat asociat construcţiei SINGLE dar înainte ca oricare din threadurile din grup să părăsească bariera. Astfel, fiecare variabilă din listă devine definită (asemeni asignării) cu valoarea aferentă din threadul ce a executat blocul structurat. *Clauza COPYPRIVATE* trimite valoarea unei variabile private tuturor thread-urilor, la sfârşitul unei directive **SINGLE,** fiind astfel cea mai folositoare pentru citirea în variabile private. Sintaxa: **#pragma omp single copyprivate(***list***)**

3.2.6. Legarea şi imbricarea directivelor

Directivele PARALLEL permit crearea concurentă de threaduri, iar directivele FOR şi SECTIONS permit distribuirea taskurilor threadurilor create. În cazul în care nu există directiva PARALLEL, secvenţa de cod aferent directivelor SECTIONS şi FOR se va executa serial.

Pentru a evita ambiguitatea referitoare la directiva de regiune paralelă la care se face referire, este nevoie de un set de reguli pentru legarea directivelor, cele mai importante sunt:

- directivele DO/FOR, SECTIONS, SINGLE, MASTER şi BARRIER se leagă de cea mai apropiată directivă PARALLEL.
- directiva ORDERED de leagă de cea mai apropiată directivă DO.

Paralelism imbricat (NESTED). Standardul oferă construcţii pentru specificarea paralelismului imbricat, ce poate fi activat cu variabila de mediu OMP_NESTED sau cu rutina OMP_SET_NESTED.

Imbricarea directivelor OpenMP respectă următoarele reguli:

- dacă este întâlnită o directivă PARALLEL în interiorul unei alte directive PARALLEL va fi creată o nouă echipă de thread-uri, însă noua echipă va conţine numai un thread, până la activarea paralelismului imbricat.
- directivele FOR, SECTIONS şi SINGLE legate la aceeaşi directivă PARALLEL nu pot fi imbricate reciproc
- directivele CRITICAL cu acelaşi nume nu pot fi imbricate reciproc.

Directive orfane. Directivele sunt active în scopul *dinamic* (de execuţie) a regiunii paralele, nu doar în scopul *lexical (sintactic)* , proprietate deosebit de utilă deoarece permite un stil modular de programare, însă care poate fi foarte confuz, dacă arborele de apeluri este complicat.

Din aceste considerente există un set de reguli suplimentare referitor la domeniul de aplicabilitate numit *scop de date* (data scope attributes), astfel atunci când se apelează o subrutină din interiorul unei regiuni paralele:

- variabilele globale şi blocurile COMMON sunt partajate, mai puţin în cazul în care sunt declarate cu THREADPRIVATE
- variabilele locale statice în C/C++ sunt partajate
- variabilele din lista de argumente moştenesc atributele cu scop de dată din rutina apelantă.
- toate celelalte variabile locale sunt private.

3.2.7.Funcţii de bibliotecă pentru sincronizare

Deoarece sunt situaţii în care este necesară mai multă flexibilitate decât cea oferită de directivele CRITICAL şi ATOMIC, pot fi utilizate lacătele.

Un lacăt (lock) este o variabilă specială care poate fi setată de un thread, astfel nici un alt thread nu mai poate seta lacătul, până în momentul în care acesta este resetat. Setarea unui lacăt poate fi blocantă sau non-blocantă, un lacăt trebuie iniţializat înainte de a fi utilizat, şi poate fi distrus când nu mai este necesar. Este posibilă utilizarea de *lacăte imbricate* ce pot fi utilizate multiplu de către acelaşi thread. Aceste funcţii permit aceluiaşi thread să seteze un lacăt de mai multe ori, înainte de a-l reseta de acelaşi număr de ori, rutinele corespunzătoare vor citi sau actualiza valoarea cea mai recentă

a variabilei, nefiind astfel necesară utilizarea explicită a directivei flush pentru a asigura consistenţa variabilei lacăt între diferite threaduri.

```
void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_init_nest_lock (omp_nest_lock_t *lock);
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t *lock);
void omp_unset_nest_lock (omp_nest_lock_t *lock);
void omp_test_nest_lock (omp_nest_lock_t *lock);
```

Pentru *analiza de performanță* şi instrumentarea codului aplicației cu scopul de a identifica timpul necesar procesării este utilă funcția *omp_get_wtime()* ce returnează valoarea timpului raportat la ceasul sistem.

Deasemenea este util Analizorul de concurență pentru Visual Studio care poate fi descărcat ca şi componentă distinctă de la adresa https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019 şi care va fi instalat pentru integrare în Visual Studio- IDE, cu scopul de a vizualiza variația gradului de paralelism pentru aplicație şi consumul de memorie la execuția codului.

3.3. Exemple

3.3.1.Implementarea unei aplicații pentru *calculul paralel a numărului Pi* prin integrare numerică folosind directive OpenMP (sursa [1])

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#define omp_get_num_threads() 1
#endif

#define NUM_THREADS 4
static long num_steps = 100000000;
double step;

int main() {
  double x;
  double pi;
  int thread_count;
  int i;

  //change in x (i.e. width of rectangle)
  step = 1.0/(double)num_steps;

  #ifdef _OPENMP
  omp_set_num_threads(NUM_THREADS);
  #endif
  #pragma omp parallel
  { int thread_id;
    double sum; //create a local sum to eliminate false sharing
    int t_count; //local copy of thread count
    double x;
    int i;
    thread_id = omp_get_thread_num();
    t_count = omp_get_num_threads();
    if (thread_id == 0) {
      thread_count = t_count;}
```

```
    //calculate the summation of F(x)
    // (i.e. sum of rectangles)
    //in the approximation of pi

    for (i=thread_id, sum = 0.0; i < num_steps; i = i+ t_count) {
      //calculate height
      x = (i+0.5)*step;
      sum = sum + 4/(1.0+x*x); //sum F(x)}
    sum = sum * step;
    #pragma omp atomic
      pi = pi + sum; //ensures calculation is atomic }
  printf("pi = %f", pi);}
```

3.3.2.Implementarea *produsului dintre o matrice şi un vector* folosind construcţiile de partajare a sarcinilor OpenMP şi serializarea acceselor la variabila totală partajată ce cumulează rezultatele. (sursa [1])

```
#include "omp.h"
#define SIZE 10
main ()
{float A[SIZE][SIZE], b[SIZE], c[SIZE], total;
int i, j, tid;
total = 0.0;
for (i=0; i < SIZE; i++)
  {
  for (j=0; j < SIZE; j++)
    A[i][j] = (j+1) * 1.0;
    b[i] = 1.0 * (i+1);
    c[i] = 0.0; }
printf("\nStarting values of matrix A and vector b:\n");
for (i=0; i < SIZE; i++)
  {printf("  A[%d]= ",i);
  for (j=0; j < SIZE; j++)
    printf("%.1f ",A[i][j]);
    printf("  b[%d]= %.1f\n",i,b[i]); }
    printf("\nResults by thread/row:\n");

/* Create a team of threads and scope variables */
#pragma omp parallel shared(A,b,c,total) private(tid,i)
  {tid = omp_get_thread_num();

/* Loop work-sharing construct - distribute rows of matrix */
#pragma omp for private(j)
  for (i=0; i < SIZE; i++)
    {for (j=0; j < SIZE; j++)
      c[i] += (A[i][j] * b[i]);

    /* Update and display of running total must be serialized */
    #pragma omp critical
      {total = total + c[i];
      printf("  thread %d did row %d\t c[%d]=%.2f\t",tid,i,i,c[i]);
      printf("Running total= %.2f\n",total);}}
/* end of parallel i loop */}
/* end of parallel construct */

printf("\nMatrix-vector total - sum of all c[] = %.2f\n\n",total);}
```

3.3.3.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului OpenMP ,exemple pe care le puteți descărca de la adresa *https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/03_OpenMP*

3.4. Întrebări teoretice

3.4.1. Ce este o barieră implicită şi cum este ea implementată în OpenMP?

3.4.2. În modelul de date OpenMP, ce reprezintă FIRSTPRIVATE şi LASTPRIVATE? Imaginați un exemplu simplu de utilizare.

3.4.2. Identificați construcțiile din modelul de programare OpenMP prin intermediul cărora se poate seta și verifica gradul de paralelism .

3.4.2. Explicați diferența în semnificația și utilizarea clauzelor SINGLE și MASTER.

3.4.3. Cum poate verificat și activat paralelismul dinamic al threadrulor, dar suportul pentru execuția unor clauze Parallel For imbricate?

3.5. Probleme propuse

3.5.1.Implementaţi un algoritm de înmulţire a două matrici pătratice folosind planificarea statică a threadurilor. Modificaţi soluţia propusă pentru înmulţirea matricilor astfel încât folosind variabila de mediu OMP_NUM_THREADS să controlaţi numărul de threaduri, analizând totodată performanţele soluţiei la modificarea acestuia. Consideraţi următoarele cazuri de analiză: paralelizarea buclei exterioare, paralelizarea ambelor bucle, paralelizarea tuturor celor trei bucle.

3.5.2.Descrieti în pseudocod o formulare paralelă a înmulțirii matrice/vector pentru care matricea este partiționată 1D pe blocuri de-a lungul coloanelor , iar vectorul este egal partiționat proceselor, pentru un sistem cu memorie partajată. Determinați timpul de execuție pentru cele două alternative de partiționare pe blocuri 1D pe linie, respectiv pe coloane.Propuneți o soluție de implementare bazată pe OpenMP, identificând cele mai potrivite funcții de bibliotecă pentru implementarea soluției.

3.5.3.Implementați o coadă multiacces pentru un set de threaduri ce inserează, respectiv extrag conținut, folosind construcții de sincronizare de tip mutex.Cuantificați timpul necesar pentru 1000 de operații de inserare și 1000 operații de extragere generate de 64 threaduri ce produc date , respectiv 64 threaduri ce consumă date din coadă.

3.5.4.Ilustrați utilizarea în manieră recursivă a lacătelor folosind un algoritm de căutare de tip arbore binar. Programul va primi ca intrare o listă mare de numere ce va fi divizată multiplelor threaduri. Fiecare thread va încerca inserarea propriilor elemente în arbore folosind un singur lacăt asociat arborelui. Demonstrați că un singur lacăt devine o gâtuire și pentru un număr moderat de threaduri.

3.6. Miniproiect

3.6.1. Să se implementeze și evalueze versiuni paralele pentru algoritmii de sortare *quick/merge/ radix* folosind modelul de programare paralelă bazat pe partajarea memoriei și biblioteca OpenMP v3.0 (se vor propune și se vor implementa algoritmi astfel încât să beneficieze de abstracțiunea task introdusă în OpenMP v3) .Pentru fiecare algoritm este necesară descrierea soluției în pseudocod a algoritmului secvențial și paralel, oferind detalii referitoare la complexitate și specificând funcțiile de bibliotecă OpenMP utilizate în implementare. Să se evalueze performanța algoritmilor pentru diferite dimensiuni ale problemei și număr de nuclee (core), calculând accelerația și eficiența obținute.Se vor genera mai multe execuții menținând unul din cei doi parametrii variabili și anume timp , respectiv dimensiunea problemei la aceeași valoare și modificându-l pe celălalt. Evaluarea se va realiza prin reprezentare grafică , importantă fiind alegerea sa corectă pentru a permite o bună interpretare a rezultatelor.

3.7. Referințe bibliografice

1. Site oficial - API și exemple :http://openmp.org/wp/2009/04/download-book-examples-and-discuss/
2. Tutorial OpenMP : https://computing.llnl.gov/tutorials/openMP/
3. Microsoft OpenMP https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019
4. Configurari Visual Studio pentru suport OpenMP: http://msdn.microsoft.com/en-us/library/fw509c3b.aspx
5. Chandra R., Parallel programming in OpenMP https://apps2.mdp.ac.id/perpustakaan/ebook/Karya%20Umum/Parallel_Programming_in_OpenMP.pdf

4.MPI – interfața pentru programarea paralelă a sistemelor cu memorie distribuită

Specificaţia *MPI (Message Passing Interface*) cunoaşte diverse implementări (API-uri MPI) sub forma unor biblioteci ce conţin un set larg (peste 250) de rutine pentru schimbul de mesaje, managementul datelor şi proceselor, rutine ce pot fi utilizate pentru un spectru variat de aplicaţii ştiinţifice implementabile eficient pe diverse sisteme paralele şi/ sau distribuite.

4.1. Obiective

- Studiul bibliotecii de programare paralelă a sistemelor cu memorie distribuită bazată pe mesaje MPI
- Procese MPI şi tipuri de comunicaţie
- Comunicatori, grupuri de procese, topologii virtuale, comunicația colectivă-mecanisme suport pentru procesare paralelă performantă
- Implementarea unor algoritmi paraleli în modelul transferului de mesaje

4.2. Concepte

Standardul MPI defineşte o *interfaţă de programare paralelă prin schimb explicit de mesaje*, ce poate fi utilizată pentru scrierea de aplicaţii pentru sisteme de tip MIMD (multiple instructions multiple data) cu memorie distribuită / partajată, respectiv pentru sisteme SIMD (Single instruction multiple data) ce presupun paralelizare prin decompoziţia datelor. MPI este o bibliotecă de funcţii, ea defineşte nume de funcţii, secvenţe de apel şi rezultatele pentru un set de funcţii care pot fi apelate în programe C, C++ şi Fortran 90. Compilarea programelor poate fi realizată cu un compilator standard, sau cu diverse alte compilatoare performante, iar în faza de editare a legăturilor este introdusă şi biblioteca MPI.

Principalele caracteristici care au determinat o largă răspândire a standardului sunt:

- oferă o interfaţă de programare pentru aplicaţii paralele/distribuite
- oferă performantă, portabilitate şi uşurinţă în folosire, semantica sa permite implementări diverse;
- semantica interfeţei este independenta de limbaj (C, C++, Fortran)
- permite comunicaţie eficientă *punct-la-punct şi punct-multipunct* în medii eterogene (reţele diverse);
- conţine un set de rutine pentru lucrul cu *grupuri de procese* , permite scheme de adresare complexe.
- permite *comunicaţie colectivă şi operaţii de calcul global* ,ce conferă scalabilitate sistemului
- oferă contexte de comunicare şi comunicatori pentru dezvoltarea de biblioteci de funcţii necesare programării paralele;
- conţine rutine pentru definirea de topologii virtuale de grup;
- oferă simplitate dar şi completitudine bazat pe un număr extrem de mare de funcţii, însă aplicaţii relativ complexe pot fi programate folosind un *set minimal de 6 funcţii esenţiale.*

4.2.1. Modelul de comunicație

Soluţia oferită de MPI constă în specificarea mesajelor la un nivel înalt de abstractizare, capabil să reflecte faptul că transmisia unui mesaj necesită o structurare mai complexă decât un simplu string de biţi, astfel structura unui mesaj devine *adresa, contor, tip_date*. Termenul de *buffer de mesaj* poate fi utilizat cu semnificaţii multiple, astfel:

- poate referi o zonă de memorie *specificată în aplicaţie* pentru stocarea datelor sau utilizatorul poate seta o zonă de memorie utilizată ca o zonă intermediară pentru memorarea unor mesaje arbitrare ce apar în aplicaţie
- poate reprezenta o zonă de memorie creată şi gestionată de *sistemul MPI*

Sincron vs asincron. Mesajul poate fi transferat direct între bufferele declarate în aplicaţia utilizator, sau pentru comunicaţie asincronă, mesajul este *stocat temporar* într-un buffer creat dinamic de sistem înaintea depozitării sale în bufferul receptor, respectiv utilizatorul poate declara iniţial un bufer suficient de mare astfel încât să permită transferul oricărui mesaj ce necesită stocare intermediară şi utilizare în operaţia de transfer. Comunicare prin transfer sincron de mesaje defineşte modul de comunicare ce poate fi regăsit în limbajul teoretic de comunicare prin transfer de mesaje al lui Hoare- CSP. Cele două procese corespunzătoare transmiterii–recepţiei de mesaje se încheie doar atunci când mesajul transmis este şi recepţionat. Avantajele pe care le prezintă referă semantica simplă şi uşor de utilizat. Un proces care revine din subrutina send va şti cu certitudine că mesajul trimis a fost recepţionat. De asemenea un avantaj important îl constituie faptul că nici sistemul şi nici utilizatorul nu trebuie să menţină un buffer pentru mesaj, acesta poate fi copiat direct în spaţiul de adrese al procesului receptor. Datorită simplităţii ,un număr mare de sisteme paralele implementează diferite variante ale acestui mod de comunicare.

Comunicarea bazată pe emisie - recepţie blocantă este caracterizată de transmiterea unui mesaj în mod blocant, executată *fără a aştepta recepţia acestuia* de către procesul destinaţie, astfel este posibilă încheierea execuţiei unei operaţii de transmisie chiar înaintea începerii execuţiei operaţiei de recepţie corespunzătoare. Tipurile de comunicare sunt prezentate in tabelul 1.

Evenimentul de comunicaţie	**Sincron**	**Blocant**	**Nonblocant**
Revenirea din send este indicată	Mesaj recepţionat	Mesaj transmis	Transmitere de mesaj iniţiată
Bufferare mesaj	Nu e necesară	E necesară	E necesară
Verificarea stării	Nu e necesară	Nu e necesară	E necesară
Încărcare generată de aşteptare	Mare	Medie	Mică
Suprapunere comunicaţie şi procesare	Nu	Da	Da

Tabel 4.1. Moduri de comunicare aflate la baza primitivelor MPI

Comunicare bazată pe emisie - recepţie neblocantă. Transmisia neblocantă se încheie *imediat după ce a fost notificat sistemul* de existenţa unui mesaj de transmis, respectiv recepţionat, mesaj care poate fi în tranzit, deja sosit sau chiar netransmis încă. Pentru această variantă, sistemul trebuie să implementeze un *buffer temporar*. Implementările performante conţin funcţii specializate pentru verificarea *stării bufferului* sau pentru implementarea unor mecanisme de aşteptare pană ce procesul poate continua (tip *wait-for).* Bufferul necesar transferului de mesaje poate fi implementat de către *sistemul de transfer al mesajelor sau mod utilizator*, la nivelul aplicaţiei.

Avantajele oferite de comunicaţia neblocantă pot fi exploatate în contextul utilizării suportului hardware de comunicaţie dedicat, astfel încărcarea indusă de comunicaţie poate fi aproape integral mascată prin operaţii neblocante. Operaţiile neblocante pot fi utilizate şi cu un protocol bufferat, astfel emiţătorul iniţiază operaţia DMA şi revine, însă datele sunt disponibile accesului (sunt sigure) doar după ce operaţia DMA s-a încheiat.

Tipuri de date MPI. O caracteristică deosebit de puternică a bibliotecii o constituie posibilitatea introducerii de argumente sub forma unor tipuri de date variate, posibil combinate, pentru toate mesajele trimise şi recepţionate. MPI asigură un set deosebit de complex de tipuri de date predefinite, set care include toate tipurile de bază din C, C++ şi Fortran, alături de două tipuri de date specifice MPI:MPI_BYTE şi MPI_PACKED.

Context de comunicaţie. Fiecare comunicare de mesaj se derulează într-un anumit context. Mesajele sunt întotdeauna primite în contextul în care au fost transmise, iar mesaje transmise în contexte diferite nu interferă.

Proces şi grup de procese. Procesul MPI este unitatea fundamentală de calcul, fiind constituit dintr-un *fir de control independent* şi un *spaţiu de adrese privat*, astfel un proces nu poate accesa direct variabilele definite în spaţiul de adrese al altui proces, comunicarea fiind bazată pe transfer de mesaje. Contextul este partajat de un grup de procese. Procesele MPI se execută fără a avea însă mecanisme de încărcare a codului pe procesoare, atribuire a proceselor procesoarelor sau mecanisme de creare şi distrugere de procese. Aceste mecanisme sunt de obicei, în cadrul implementărilor curente, *preluate de la sistemul de operare pe care rulează* procesul MPI, fapt ce poate crea dificultăţi de portare a acestor biblioteci între sisteme de operare diferite. MPI este proiectat a fi thread-safe, iar grupurile de procese definite în MPI sunt dinamice, apartenenţa la grup este statică însă suportă ataşarea-detaşarea din grup. Grupurile se pot suprapune fapt ce permite unui proces să fi membru al mai multor grupuri simultan. Procesele sunt identificate în cadrul grupului printr-un întreg (rank), care ia valori între 0 şi n-1, unde n este numărul proceselor din grup. Iniţializarea unei aplicaţii MPI folosind rutina de iniţializare MPI_INIT, creează un singur grup, numit MPI_COMM_WORLD, care include toate procesele MPI ce formează aplicaţia.

Procesele pot utiliza *comunicaţie punct-la-punct* cu scopul transmisiei- recepţionării de mesaje utilă implementării comunicaţiilor locale nestructurate. De asemenea un grup de procese poate apela operaţii de tip *comunicaţii colective* pentru a simplifica implementarea unor operaţii globale, deşi standardul nu oferă în mod explicit suport pentru multithreading, unele implementări includ mecanisme de acest fel.

Comunicatorul integrează informaţiile de context şi informaţiile de identificare a grupului. MPI_INIT defineşte pentru fiecare proces apelat comunicatorul implicit MPI_COMM_WORLD. Toate comunicaţiile MPI necesită un comunicator ca argument, iar procesele MPI nu pot comunica decât dacă *partajează* un comunicator. Fiecare comunicator identifică un grup reprezentând o listă de procese, fiecare proces este numerotat, identificatorul fiecăruia numindu-se rang (*rank*). Acest rang identifică în mod unic un proces şi poate fi folosit pentru specificarea sursei sau destinaţiei unui mesaj. Folosind MPI_COMM_WORLD, fiecare proces poate comunica cu oricare altul, iar grupul MPI_COMM_WORLD reprezintă mulţimea tuturor proceselor MPI. Informaţiile specifice comunicatorului pot fi accesibile prin funcţiile:

MPI_COMM_RANK (MPI_comm comm, int *rank)
Funcţia MPI_COMM_RANK returnează rangul procesului apelant în grupul comunicatorului *comm*.
MPI_COMM_SIZE (MPI_comm comm, int *size)
Funcţia MPI_COMM_SIZE returnează în variabila *size* numărul de procese din grupul asociat comunicatorului *comm*. Fiecare proces aparţinând unui comunicatori este identificat prin rangul său.

4.2.2. Modelul computaţional

Stilul de programare care poate fi folosit cu MPI se bazează fie pe modelul **MIMD** (Multiple instructions multiple data) ce presupune existenţa de programe diferite pentru fiecare procesor, fie pe modelul **SPMD** (Single Program Multiple Data), caz în care există un singur program, astfel procesele MPI provin din acelaşi program, dar execută porţiuni diferite de cod selectate pe baza unor instrucţiuni de condiţie. Standardul MPI a fost proiectat astfel încât programatorul începător să nu simtă complexitatea de la primele programe. Programarea unei aplicaţii minimale MPI se poate face

cunoscând doar un set minimal de 6 funcţii ce reprezintă nucleul suficient pentru scrierea unei aplicaţii:

MPI_Init (int *argc, char**argv) este funcţia care iniţializează variabilele locale MPI şi înştiinţează sistemul global MPI despre execuţia unui nou proces MPI. Funcţia acţionează şi ca barieră de iniţializare, în sensul ca nici un proces MPI nu-şi poate continua execuţia până când toate celelalte procese nu au executat partea lor de iniţializare, argc/argv sunt argumentele la nivel de linie de comanda pentru programul C.
MPI_Comm_rank(), returnează identificatorul procesului MPI în cadrul grupului specificat în lista de parametri.
MPI_Comm_size**() î**ntoarce numărul de procese care fac parte din grupul specificat ca parametru
MPI_Send() şi MPI_Recv() sunt funcţiile de bază care implementează comunicaţia prin mesaje între procese**.**
MPI_Finalize() pentru anunţarea sistemului MPI despre terminarea procesului MPI cu succes (valoarea lui ierr=0), această rutină este apelată de fiecare proces MPI. După această funcţie nici o altă funcţie nu poate fi apelată, toate comunicaţiile în aşteptare trebuie încheiate înaintea apelului ei.

Apelul funcţiilor de bibliotecă necesită includerea antetului **mpi.h** ce conţine constantele referite în rutinele MPI (tipuri de date, comunicatori, erori) şi interfeţele funcţiilor MPI. O operaţie este considerată *terminată local* în cadrul unui anumit proces, dacă acesta a terminat partea sa ,atât în comunicaţia punct-la-punct, cât şi în cea colectivă. O operaţie de comunicare este *terminată global*, dacă toate procesele implicate în comunicare au *terminat local* părţile ce le revin din aceasta.

4.2.3. Modul de comunicaţie punct-la-punct

O aplicaţie poate reduce latenţa de comunicare presupunând că platformele hardware pe care este realizată implementarea includ mecanisme care permit *rutarea şi gestiunea mesajelor paralel cu alte procesări*. Biblioteca MPI garantează respectarea următorului set de proprietăţi (reguli semantice): pentru comunicarea punct la punct, astfel: garantarea preluării mesajelor în ordinea în care au fost trimise, perechile send şi receive nu pot rămâne în permanenţă nerezolvate (necorelate).
Mesajele transmise reprezintă o secvenţă de articole de acelaşi tip, identificarea mesajelor realizându-se prin identificatorul **sursei (rank) şi o etichetă (tag)** ataşată mesajului, valori interpretate relativ, în cadrul zonei de comunicaţie identificată de un anumit comunicator. Există situaţii în care este necesar ca recepţia să fie efectuată indiferent de sursă (MPI_ANY_SOURCE) sau de etichetă (MPY_ANY_TAG).

Programele paralele necesită implementarea unor soluţii algoritmice capabile să minimizeze timpul de răspuns, iar MPI oferă propriile mecanisme pentru instrumentarea codului astfel încât să permită analiza performanţei. Funcţia **MPI_Wtime**, returnează o valoare dublă precizie a numărului de secunde raportat la un anumit moment de timp trecut, garantat nemodificabil pe durata execuţiei programului, fapt ce permite inserarea diverselor apeluri ale rutinei în codul sursă pentru măsurarea unor intervale de timp de execuţie. Funcţiile bibliotecii MPI utilizate în analiza performanţelor bazat pe elementul timp de procesare sunt **MPI_Wtime (void), MPI_Wtick(void).** Lista parametrilor utilizaţi în rutinele MPI şi semnificaţia lor,respectiv lista codurilor de eroare sunt prezentate în tabelul 4.2.:

ierror	**Cod de eroare MPI**
Comm	Identificarea comunicatorului utilizat în transmisia mesajului
Rank	Rangul procesului apelant în grup sau comunicator
buffer	Adresa de start a buferului ce conţine mesajul(transmis-receptionat)
count	Număr de articole specificate în bufer (număr maxim de articole receopţionate)
datatype	Tipul de dată a fiecărui element transmis

parametru	Semnificaţie
Root	Rangul unui proces specific (sursa sau destinaţia pot fi identice)
Size	Numărul de procese din grup
Tag	Tagul de mesaj transmis-recepţionat
source	Rangul procesului sursă în comunicator
dest	Rangul procesului destinaţie în comunicator
status	Tablou de întregi ce conţine starea, referă o modul de recepţie a mesajului (sursa, tagul şi codul de eroare)

Tabel 4.2. Coduri de eroare şi parametrii rutinelor MPI

Pentru a putea construi o aplicaţie paralelă performantă este necesar un control performant asupra transmiterii mesajelor. În acest sens, MPI defineşte patru moduri de comunicare: **standard, sincron, bufferat şi ready** (vezi tabelul 4.3.). Funcţia de comunicare se obţine prin combinarea modului de comunicaţie cu un mod de blocare, moduri ce vor fi detaliate la secţiunea funcţii pentru comunicaţie. Toate modurile de comunicaţie există atât în forma blocantă cât şi neblocantă, în forma blocantă, întoarcerea dintr-o funcţie implică terminarea ei cu succes MPI nu impune restricţii asupra modului de potrivire a funcţiilor de transmisie cu cele de recepţie, existând astfel mai multe posibile combinaţii a căror funcţionare corectă intră însă în sarcina programatorului. MPI defineşte două moduri de comunicare:

- fără blocare (imediat), caz în care funcţiile de transmisie sau recepţie se termină înainte ca informaţia să fie trimisă/recepţionată, iar bufferul indicat într-o funcţie neblocantă nu trebuie folosit până la detectarea sfârşitului funcţiei iniţiate.
- cu blocare, caz în care după terminarea funcţiei, bufferul de transmisie sau recepţie poate fi refolosit.

Tip	Descriere
Send sincron	operaţia de trimitere poate începe oricând, dar nu se poate termina decât atunci când mesajul a ajuns la receptor
Send buferat	utilizatorul poate furniza sistemului un buffer pentru a permite ca operaţia de trimitere să se termine întotdeauna înainte ca mesajul să fie recepţionat la destinaţie.
Send standard	operaţia de trimitere poate începe chiar dacă operaţia corespunzătoare de recepţie nu a început.
Send ready	operaţia send poate începe doar dacă operaţia corespunzătoare de recepţie a început.
Receive	Se completează la sosirea unui mesaj

Tabel 4.3. Moduri de comunicare MPI

Sintaxa rutinelor pentru transmisia mesajelor este:
MPI_Send(void*buffer, int count, MPI_datatype datatype, int dest, int tag, MPI_comm comm)

MPI_Recv(void* buffer, int count, MPI_datatype datatype, int source, int tag, MPI_comm comm, MPI_status status)

După ce mesajul a fost transmis, variabila Status poate fi utilizată pentru a colecta informaţii referitoare la operaţia MPI_recv, aceasta este o structură de date cu trei câmpuri: sursa mesajului, tagul şi cod de eroare, iar lungimea mesajului transmis poate fi verificată cu ajutorul funcţiei MPI_get_count.

Transmisie blocantă standard (Standard send). Forma *Standard send* se încheie odată ce mesajul a fost trimis, mesaj care poate să ajungă sau nu la destinaţie, iar ignorarea acestui fapt poate duce la comportări inconsistente ale programului. Sintaxa:

int MPI_SEND (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

unde:

buffer este adresa datei ce va fi trimise;
count este numărul de elemente de tipul *datatype* conţinute de *buffer*
dest reprezintă procesul destinaţie specificată prin intermediul rangului din cadrul comunicatorului *comm*;
tag este un marcator la dispoziţia programatorului pentru a distinge între diferite tipuri de mesaje;

Transmisie sincronă (Synchronous send). *Synchronous send* se foloseşte atunci când se doreşte confirmarea primirii mesajului, astfel destinaţia trimite spre sursă o confirmare iar transmisia este considerată încheiată numai după primirea confirmării. Primitiva send se consideră încheiată atunci când o operaţie receive a fost lansată, astfel bufferul aplicaţiei este disponibil pentru a recepţiona mesajul. Este posibilă o transmisie sincronă neblocantă ce nu presupune transmiterea mesajului, ci doar inexistenţa bufferării suplimentare la receptor, dar un buffer sistem la emiţător poate fi necesar, iar pentru a elimina copieri multiple ale mesajului se poate utiliza emisia blocantă. Această formă de transfer se foloseşte atunci când este nevoie de siguranţă a transmisiei, pentru a oferi comportament determinist. Sintaxa:

int MPI_SSEND (void * buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Transmisie buferată (Buffered send). Forma *Buffered send* foloseşte un buffer pentru trimiterea mesajului, dacă acesta nu poate fi trimis la momentul dorit. Avantajul faţă de metoda clasică îl constituie predictibilitatea recepţiei mesajului. Ataşarea unui buffer se realizează cu ajutorul funcţiei:

int MPI_BUFFER_ATTACH (void *buffer, int size)

Numai un singur buffer poate fi ataşat unui proces la un moment dat. Detaşarea se realizează cu:

int MPI_BUFFER_DETACH (void *buffer, int size)

Observaţii:

- după detaşare, utilizatorul poate reutiliza sau dealoca spaţiul ocupat de tampon
- unele operaţii, de ataşare şi detaşare, au un argument de tip void*; ele sunt folosite diferit – un pointer la tampon este transmis la *attach*; adresa pointerului este utilizată la *detach,* astfel că acest apel întoarce valoarea pointerului. Argumentele sunt definite ambele ca void* (şi nu void* respectiv void **) pentru a evita conversii forţate de tip (cast).

Transmisie ready (Ready send). Formele *Ready send*, ca şi *buffered send* se termină imediat, iar comunicarea este garantată a se efectua cu succes dacă o operaţie *receive* corespunzătoare a fost apelată. La expedierea mesajului, procesul expeditor pune mesajul în comunicator, sperând ca destinatarul să îl aştepte şi să îl primească. Dacă acesta nu îl acceptă, mesajul poate fi abandonat sau se generează o eroare.

int MPI_RSEND (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Recepţie blocantă (Blocking receive). Sintaxa:

int MPI_RECV (void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status status)

unde:

- *source* este rangul expeditorului în grupul asociat comunicatorului *comm*; ca şi sursă se poate specifica MPI_ANY_SOURCE, ceea ce este echivalent cu orice sursă;
- *tag* specifică tipul mesajelor ce vor fi acceptate, se poate de asemenea specifica MPI_ANY_TAG ca fiind identificator universal de mesaj.

Informaţia este returnată de către funcţia MPI_RECV în variabila *status*. Acest argument poate fi testat direct pentru a afla sursa mesajului, dimensiunea şi identificatorul de mesaj (de exemplu poate fi necesară identificarea sursei unui mesaj recepţionat folosind MPI_ANY_SOURCE).

4.2.4. Comunicaţii colective,comunicatori şi topologii virtuale

Mecanismele abstracte folosite în comunicaţie sunt : comunicatori, context de comunicare, topologii virtuale.În multe aplicaţii paralele este necesar ca operaţiile de comunicaţie să fie restricţionate la un set limitat de procese. MPI defineşte o serie de concepte şi introduce diverse mecanisme pentru lucrul cu procese structurate în grupuri şi organizate folosind comunicatorii. MPI introduce noţiunea de *comunicator* pentru a defini cadrul unei comunicaţii colective, în particular procesele care sunt implicate în această comunicare, dar şi pentru a permite construcţia modulară a programelor paralele în mod: secvenţial, paralel sau-şi concurent.

Un comunicator defineşte un *context de comunicare alături de o colecţie ordonată de procese* implicate într-un subset al calculului de efectuat. Prin introducerea comunicatorilor, comunicaţia între subseturi de procese ale programului poate fi structurată. Comunicaţia între procese din grupuri distincte este posibilă prin definirea unui *inter-comunicator*.

Comunicaţia inter-grupuri este, de obicei, gestionată de un conducător al grupului. Grupurile pot fi create şi prin intermediul reuniunii, intersecţiei, diferenţei între grupuri deja existe. Noţiunea de comunicator este fundamentală pentru definirea într-un mod modular a bibliotecilor de funcţii care pot fi invocate de procesele utilizator. Putându-şi defini propriul comunicator (alocat de către sistem, cu asigurarea unicităţii), procesele sistem pot schimba mesaje fără a interfera cu comunicaţia dintre procesele aplicaţiei utilizatorului.

Este necesară distincţia clară între diferitele tipuri de comunicatori. Astfel un *comunicator* este un obiect cu atribute şi reguli pentru creare, utilizare şi distrugere. Comunicatorul specifică un domeniu de comunicare ce poate fi utilizat de comunicarea punct la punct sau colectivă

- Un *intracomunicator* este utilizat pentru comunicarea în interiorul unui grup (comunicare intra-grup); el are două atribute: *grupul de procese şi topologia grupului.*
- Un *intercomunicator* este utilizat pentru comunicarea punct la punct între două grupuri diferite (comunicare inter-grup); atributele lui sunt cele două grupuri, fără a specifica topologia.
- Un *domeniu de comunicare* este un set de comunicatori. Dacă acest domeniu este pentru comunicare intra-grup atunci toţi comunicatorii sunt intracomunicatori şi au aceleaşi atribute.

Unui comunicator i se pot ataşa informaţii adiţionale alături de cele de grup şi context, informaţie numită **topologie.** Topologia reprezintă un mecanism ce realizează asocierea diverselor *scheme de adresare proceselor* aflate într-un grup. Grupurile şi comunicatorii sunt obiecte opace, detaliile reprezentării lor interne sunt specifice implementării MPI, iar accesarea lor este realizată folosind un handle. Contextele nu sunt utilizate explicit în funcţiile MPI, ci sunt asociate în mod *implicit* grupurilor de procese la crearea lor.

Procesele unui grup au ranguri de la 0 la n-1 ,n fiind numărul de procese din grup. În multe aplicaţii, ordonarea liniară a proceselor, după rang, nu reflectă tiparul de comunicare între procese.

Adesea, procesele trebuie aranjate în topologii cu două sau mai multe dimensiuni. În cazul general, tiparul de comunicare între procese corespunde unui graf. Un astfel de aranjament al proceselor, care reflectă comunicările punct la punct dintre ele reprezintă topologia virtuală a grupului de procese. Trebuie făcută distincţie între topologia virtuală a proceselor şi topologia reală a sistemului pe care acestea sunt executate. Topologia virtuală poate fi exploatată de sistem în plasarea proceselor pe procesoare, pentru a ameliora performanţele.

Trebuie reţinut faptul că topologia virtuală reflectă caracteristicile aplicaţiei şi poate fi folosită în *îmbunătăţirea performantelor.* Sunt cunoscute tehnici standard de mapare a unor topologii grilă / tor pe hipercuburi sau grile de procesoare. Pentru topologiile graf metodele sunt mai complexe şi se bazează adesea pe euristici. În afara informaţiei furnizate algoritmului de plasare a proceselor pe procesoare, topologiile virtuale au ca rol facilitarea scrierii programelor, într-o formă mai uşor de înţeles. Topologia virtuală poate fi considerată ca un atribut deosebit de puternic al unui comunicator, atribut ce permite descrierea structurii de interconectare a proceselor dintr-un grup, ea este folosită pentru a mapa procesele pe arhitectura fizică.

Topologia poate fi definită prin grupul de procese, numărul de conexiuni al fiecărui proces şi partenerul pentru fiecare conexiune Topologia virtuală este distinctă de graful programului, pentru că două procese pot comunica chiar dacă în topologie nu sunt conectate. Topologia virtuală descrie acei comunicatori care trebuie consideraţi de maparea grupului de procese pe arhitectura fizică. Noţiunea de topologie virtuala este deosebit de utilă, atunci când nu este disponibil un mecanism eficient de mapare a topologiei virtuale pe un calculatomasiv paralel. Astfel, dacă procesele utilizatorului vor comunica în principal cu cei mai apropiaţi vecini după modelul unui grid bidimensional, se va putea crea o topologie virtuală care să reflecte acest fapt. In acest mod este realizat accesul la rutinele convenabile, de exemplu, calculând rangul oricărui proces dat prin coordonatele sale în grid, luând în considerare condiţiile limită şi returnând MPI_NULL_PROC dacă se iese din grid. În particular există rutine care calculează rangurile celor mai apropiaţi vecini, rang ce poate fi folosit ca argument pentru primitivele de transmisie /recepţie MPI_SEND, MPI_RECV, MPI_SENDRECV etc. Deşi o topologie virtuală scoate în evidenţă şabloanele utilizate în comunicare într-un comunicator pentru o "conexiune", orice proces din comunicator poate comunica cu oricare altul.

O topologie virtuală poate fi modelată ca un graf în care nodurile reprezintă procese, iar arcele perechile de procese care comunică. Nu trebuie ca o comunicare să fie precedată de o deschidere explicită de canal. Ca urmare, absenţa unui arc între două noduri din graf nu înseamnă că procesele respective nu vor putea comunica între ele, aci că o astfel de comunicare nu contează la maparea topologiei virtuale de procese pe o topologie reală de procesoare. Arcele grafului de comunicare nu sunt ponderate. Deşi specificarea topologiei în termeni de graf este foarte generală, în multe aplicaţii specificarea unei topologii mai simple, regulate, este mai convenabilă. De exemplu inel, grile bi- sau tri-dimensionale, tor. Aceste topologii sunt complet definite prin numărul de dimensiuni şi prin numărul de procese pe fiecare axă de coordonate. Maparea unor topologii grilă sau tor este mai simplă decât maparea unui graf, motivând astfel interesul MPI pentru tratarea separată, explicită a acestora.

MPI permite o serie largă de topologii virtuale între care cele mai uzuale sunt topologiile de tip graf virtual, topologiile carteziene, s. a. Două noduri în graf sunt conectate dacă pot comunica. Grafurile de procese pot fi utilizate pentru a specifica orice topologie, cele mai utilizate fiind gridurile. O topologie este cartezianã dacă fiecare proces este conectat cu vecinii săi printr-un grid virtual.Mecanismul utilizat de MPI pentru a asigna ranguri proceselor dintr-un anumit domeniu de comunicaţie nu utilizează informaţie despre reţeaua de comunicaţie, ci oferă un set de funcţii ce permit programatorului *aranjarea proceselor în diverse topologii fără a specifica explicit maparea lor la procesoare*, responsabilitatea unei mapări capabile sa reducă costurile transmisiei de mesaje este în totalitate a bibliotecii MPI.

Comunicaţia colectivă permite difuzarea datelor către toate procesele unui grup, sincronizarea la bariere ca mecanism de control al execuţiei, distribuţia colectarea datelor de la procesele unui grup, operaţii globale de reducţie pe grupuri de procese (suma, minim, maxim) sau comunicaţie de tip all-to-all. Toate aceste operaţii se pot împărţi în două mari clase: *operaţii de transfer de date şi operaţii de calcul global.*

Deşi operaţiile de comunicaţie colectivă nu acţionează asemeni unei bariere ele realizează o *sincronizare virtuală*, iar programul necesită o semantică corectă astfel încât să se manifeste determinist chiar dacă este necesară includerea unor operaţii de sincronizare globală înainte respectiv după apelul funcţiei colective. Deoarece operaţiile colective sunt virtual sincrone, ele *nu necesită marcajul de mesaj*(tag-ul). Pentru cele mai multe operaţii colective există două variante astfel: transfer de date de aceeaşi dimensiune, respectiv transfer de date de dimensiuni diferite, funcție ce ce tip și volum de date necesită transfer.

Rutinele de transfer de date referă *difuzarea datelor (*broadcast), *distribuirea* (scatter) şi *colectarea rezultatelor* (gather) spre sau de la procesele unui grup. Colectarea de rezultate se poate realiza de către *toate* procesele (de ex. funcția MPI all-gather) sau de către *un singur* proces. De asemenea, există operaţii *all-to-all*, în care fiecare proces trimite date, respectiv recepţionează date de la celelalte procese, prin utilizarea unei singure funcții cu semnătură complexă.

Operaţiile de calcul global pot fi împărţite în operaţiile de reducere şi operaţii de scanare. Pentru operaţiile de reducere funcția de reducere se aplică pe toate datele corespunzătoare fiecărui proces din grup, iar pentru operaţiile de scanare, funcția de reducere se aplică pe datele proceselor cu rang mai mic decât cel al procesului care execută operația.

Funcţia de reducere trebuie să fie asociativă şi comutativă, iar rezultatul unei operaţii globale poate fi cunoscut de *toate procesele sau doar de un singur proces*. În operaţiile globale pot fi folosite atât operaţiile predefinite în MPI cât şi operaţii definite de utilizator. În cadrul comunicaţiei colective, nu se folosesc etichete de mesaj, coordonarea comunicaţiei proceselor este implicită şi toate rutinele de comunicaţie colectivă se blochează până când comunicaţia este terminată local.

Operaţii colective. Comunicarea colectivă implică un grup de procese între membrii căruia se desfăşoară comunicaţia. Biblioteca MPI suportă comunicaţii colective cum ar fi difuzarea, *dispersarea/adunarea (scatter/gather), schimb total de date, agregare şi bariera*. Pentru toate formele de comunicare colectivă, toate procesele unui grup trebuie să realizeze apelul corespunzător, cu argumente compatibile. Orice omisiune în acest sens constituie o eroare.

Caracteristici ale comunicaţiilor colective:

- comunicaţiile colective nu pot interfera cu comunicaţiile punct la punct şi vice-versa;
- comunicaţie *colectivă poate sau nu să realizeze sincronizarea* proceselor participante;
- comunicaţiile colective sunt blocante;
- *toate procesele participante într-un comunicator trebuie să apeleze comunicaţiile colective*;
- mesajul transmis este un vector cu elemente de un anumit tip
- tipul de date trebuie să fie acelaşi pentru transmisie şi recepţie

Sincronizarea la barieră este cea mai simplă operaţie de comunicare colectivă care nu implică transfer de date. Forma sa este: **MPI_BARRIER (MPI_comm comm)**

Unicul argument este comunicatorul *comm* ce defineşte grupul de procese sincronizate, astfel procesul apelant se blochează până când toţi membrii grupului execută apelul funcţiei (ating bariera).

Transmisia broadcast. In transmisia *one –to-all*, procesul root transmite acelaşi mesaj stocat în bufferul *buffer* (un număr de *count* de intrări de date având tipul *datatype)* tuturor proceselor inclusiv lui însuşi, procese incluse în comunicatorul comm. Toate procesele implicate în comunicaţie trebuie să specifice aceeaşi rădăcină (root, procesul cu identificatorul 0 de regulă).

int MPI_BCAST (void *buffer, int count, MPI_datatype datatype, int source, MPI_comm comm)

- *buffer* este adresa de început a buffer-ului, *count* este numărul intrărilor în buffer
- *datatype* este tipul datelor din buffer, source (poate fi cazul particular root) este rangul procesului sursă, iar *comm* este comunicatorul.

Conţinutul mesajului este identificat de tripletul *buffer, count, datatype,* pentru procesul sursă acest triplet specifică bufferele de emisie şi recepţie, iar pentru celelalte procese buferul de recepţie. Volumul de date transferat de procesul sursă trebuie să fie identic cu volumul de date recepţionat de fiecare proces.

Funcţia Gather. Procesul ţintă (poate fi chiar rădăcina pentru topologii de tip arbore) recepţionează mesaje personalizate de la fiecare din cele n procese (inclusiv de la el însuşi). Cele n mesaje sunt *concatenate în ordinea rangurilor proceselor* şi stocate în bufferul de recepţie al procesului receptor. Fiecare buffer emiţător este identificat de tripletul *sendbuf, sendcount, sendtype,* iar bufferul de recepţie este ignorat pentru procesele nonroot, iar pentru procesul root este identificat de tripletul *recvbuf, recvcount, recvtype*.

Funcţia implementează comunicaţia de tip *all-to-one,* toate procesele transferă datele aflate în *inbuf* procesului rădăcină, care plasează datele în locaţii contigue nesuprapuse în buferul *outbuf.*

int MPI_GATHER (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int target, MPI_comm comm)

unde:

- *sendbuf* este adresa de start a send buffer
- *sendcount* este numărul elementelor din send buffer
- *sendtype* este tipul datelor din send buffer
- *recvbuf* este adresa lui receive buffer
- *recvcount* este numărul elementelor pentru fiecare recepţie
- *target* este rangul procesului receptor

Funcţia Scatter descrie operaţia inversă operaţiei GATHER. Sintaxa sa este:

int MPI_SCATTER (void *sendbuf, int sendcount, MPI_Datatype sendtype, void recvbuf, int recvcount, MPI_Datatype recvtype, source, MPI_comm comm).

Pentru lista completă a funcţiilor de comunicaţie colectivă se vor consulta resursele [1][3].

Pentru *analiza de performanţă* şi instrumentarea codului aplicaţiei cu scopul de a identifica timpul necesar procesării este utilă funcţia *MPI_wtime()* ce returnează valoarea timpului raportat la ceasul sistem.Deasemenea este util Analizorul de concurenţă pentru Visual Studio care poate fi descărcat ca şi componentă distinctă de la adresa https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019 şi care va fi instalat pentru integrare în Visual Studio- IDE, cu scopul de a vizualiza variaţia gradului de paralelism pentru fiecare aplicaţie şi consumul de memorie la execuţia codului.

4.3.Exemple

4.3.1. Realizaţi o transformare a programului secvenţial pentru calculul lui PI bazat pe integrarea numerică a funcţiei f(x)=4/(1-x*x) în intervalul [0, 1]. Să se ofere o soluţie de paralelizare prin împărţirea intervalului de integrare aferent numărului de procesoare din sistem. Pentru algoritm se poate folosi aproximarea funcţiei cu sumarea k=1.. n (4/(1+((k-. 5)/n)**2) şi fiecare nod va primi numărul de dreptunghiuri folosite în aproximare, va calcula aria dreptunghiului şi apoi se sincronizează pentru calculul sumei globale. Implementaţi un program pentru calculul valorii lui PI ce poate rula pe un sistem cu un număr arbitrar de procesoare, astfel va trebui ca indiferent de numărul de procesoare, valoarea calculată să fie aceeaşi (sursa [5]).

```
   /* Calculates the PI. It is the area of f(x)=4/(1+x^2) in interval from 0 to 1.The
interval in splitted into n intervals which are distributed equaly to all  available
processes. The result consists in a sum of areas calculated by each processor !!! */
```

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

/* Function definition */
double f( double a ) {
    return (4.0 / (1.0 + a*a));}

int main( int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i;
    double PI25DT=3.14159265358979238462643;
    double mypi, pi, h, sum, x;
    double startwtime=0.0, endwtime;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    /* Initialize MPI */
    MPI_Init(&argc,&argv);

    /* Get no. of processes */
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

    /* Get current process ID */
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    /* Get current processor name */
    MPI_Get_processor_name(processor_name,&namelen);

    /* Print info about MPI WORLD and about current process */
    printf("MPI_WORLD has %d processes\n", numprocs);
    printf("Process %d on %s\n",myid, processor_name);

    /* Initialize no. of intervals to 0 */
    n = 0;
    while (!done) {
        /* What root process does: */
   if (myid==0) {

            /* Read no. of intervals prom keyboard */
       printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);

              /* Next line should only be used if you are too lazy to enter no.
       of intervals */
       /* if (n==0) n=100; else n=0; */

       /* Get current time */
       startwtime = MPI_Wtime(); }

   /* Send no. of intervals to all processes. Only root sends the message
   and the others receive the message */
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

   /* In no intervals defined then stop program */
   if (n==0)
            done=1;
        /* If intervals defined */
   else {
            /* Calculate f(x) where x=1/n*(i-0.5)*/
       h=1.0/(double)n;
            sum=0.0;
            for (i=myid+1;i<=n;i+=numprocs) {
                x=h*((double)i-0.5);
                sum+=f(x);}

       /* Area is 1/n*f(x) */
            mypi=h*sum;
```

```
     /* Send calculated area to root through a SUM function. I don't
    know yet if root sends it's value, but it's for sure he receives
    as result the SUM of all numbers from the other processes including
    his own. */
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

     /* What root does */
    if (myid==0)
    {
          /* Print the result of PI*/
       printf("pi is approximately %.16f, Error is %.16f\n",
                  pi, fabs(pi - PI25DT));

       /* Get current time and print total time needed */
       endwtime = MPI_Wtime();
       printf("wall clock time = %f\n",
              endwtime-startwtime);}}}

/* Terminate MPI */
MPI_Finalize();
/* Success */
return 0;
```

4.3.2.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului MPI, exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/04_MPI

4.4.Întrebări teoretice

4.4.1.Ce tipuri de comunicație oferă interfața MPI? Explicați-le succint semnificația și aplicabilitatea.

4.4.2.Identificați minim 3 tipuri de operații de tip SEND și ilustrați modul de utilizare.

4.4.3.Care sunt funcțiile MPI pentru inițializarea/închiderea contextului de execuție

4.4.4.Ce rol joacă un comunicator?Dar un grup de procese?

4.4.5.Ce proces joacă un rol distinct în structurarea și execuția codului? Explicați prin exemplificare cu o secvență de cod ,posibila execuție paralelă pe o mașină cu memorie distribuită ce dispune de mai multe procesoare.

4.5.Probleme propuse

4.5.1.Să se implementeze o aplicaţie simplă folosind transmiterea de mesaje în accepţiunea în care se definesc un proces receptor şi n-1 procese emiţător. Acestea îşi transmit identificatorul de proces şi numele hostului procesului receptor care le tipăreşte.

4.5.2. Să se implementeze un program în care sunt definite un număr impar de procese. Procesul zero iniţializează un tablou de întregi şi distribuie tabloul tuturor proceselor folosind primitiva Scatter. Procesele receptor îşi primesc părţile corespunzătoare din tabloul sursă. Fiecare proces răspunde cu un mesaj ce conţine numele hostului pe care procesul se execută şi partea sa din tabloul distribuit anterior, iar procesul rădăcină recepţionează şi tipăreşte mesajele.

4.5.3.Să se calculeze produsul a doi vectori (x, y de dimensiune n) distribuiţi unui grup de procese. Tablourile ce conţin vectorii sunt alocate static, iar dimensiunea vectorilor este divizibilă cu p numărul de procesoare din sistem. Propuneţi două soluţii de implementare pentru operaţia de reducţie şi analizaţi modul de execuţie şi performanţa folosind analizorul de concurenţă ce poate fi descărcat de la adresa http://msdn.microsoft.com/en-us/library/dd537632.aspx şi integrat în Visual Studio.

4.5.4.Implementaţi un program care să determine timpul necesar execuţiei operaţiei MPI_Barrier. Utilizaţi comunicatorul implicit MPI_COMM_WORLD. Tipăriţi dimensiunea comunicatorului

asigurându-vă ca atât emiţătorul cât şi receptorul sunt pregătiţi la începutul testului. Cum variază performanţa primitivei variind dimensiunea comunicatorului?

4.5.5.Propuneţi un program care sa realizeze recepţia corectă a mesajelor de la toate procesele emiţătoare, astfel încât toate cele 100 procese să transmită procesului 0, utilizând recepţia nonblocantă şi primitiva Wait_some.

4.5.6.Modificaţi problema conectării procesoarelor unei maşini paralele într-o topologie de tip inel înlocuind referirea proceselor vecine folosind rangurile rank-1 şi rank+1 folosind rutinele MPI specific topologice (topologii catreziene, liniare) utile pentru " lumi" de procese ce necesită comunicatori distincti.

4.6. Miniproiect

Sa se descrie în pseudocod şi să se implementeze şi evalueze performanța implementării pentru soluții paralele ale algoritmilor de înmulțire matrici optimizati pentru implementare paralelă folosind transferul de mesaje. (algoritmul Fox (https://www.cs.usfca.edu/~peter/ipp/) și algoritmul Cannon (https://bit.ly/2DdmHgr). Pentru dezvoltarea aplicașiei este acceptată orice implementare a bibliotecii din cele existente (MS-MPI, OpenMPI, MPICH2).

Deasemenea, se vor reprezenta grafic evaluările de performanță (Accelerarea S și Eficiența E) realizând un set de execuții consecutive și modificând corespunzător numărul de procese și/sau numărul de procesoare. Se vor analiza rezultatele obținute și se va evalua scalabilitatea.

4.7. Referințe bibliografice

1. API MPI https://www.mpich.org/static/docs/v3.2/www3/index.htm (similar MS-MPI)
2. Standard MPI https://www.mpi-forum.org/docs/
3. Tutorial MPI https://computing.llnl.gov/tutorials/mpi/
4. Implementări diverse :

   Microsoft MPI v6 https://www.microsoft.com/en-us/download/details.aspx?id=47259

   Compilare si executie
   http://blogs.technet.com/b/windowshpc/archive/2015/02/02/how-to-compile-and-run-a-simple-ms-mpi-program.aspx
   MPI project template (VS2015)
   https://visualstudiogallery.msdn.microsoft.com/90fb60d4-0b8c-472f-8135-683d3c45f45a

   OpenMPI : http://www.open-mpi.org/
   Instructiuni de instalare
   http://programmers-journal.blogspot.ro/2013/09/netbeans-ide-configuration-for-openmpi.html

   MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich2
   Instructiuni de instalare
   http://yazid.blog.umt.edu.my/files/2010/09/05-2-MPICH_VS2008manual.pdf

5. Programare paralelă- manual open source Universitatea Princeton :
   https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf

5.CUDA-programare masiv paralelă folosind procesoare grafice

CUDA este o platformă de calcul paralel şi o interfaţă de programare a aplicaţiilor dezvoltată cu scopul de a accelera operaţiile de calcul masiv pe date multiple prin folosirea puterii de calcul disponibilă în procesoarele grafice GPU. Această platformă permite dezvoltatorilor şi inginerilor de software să utilizeze o unitate de procesare grafică cu procesor CUDA pentru procesarea generală, abordare numită GPGPU (engl.General-Purpose computing on Graphics Processing Units). Termenul GPGPU denotă un procesor grafic cu o flexibilitate ridicată de programare, capabil de a rezolva şi probleme generale de procesare. In executie, o arhitectură de tip GPU foloseste paradigma SIMD (single instruction multiple data) din taxonomia Flynn, ceea ce presupune schimb rapid de context între thread-uri, planificarea în grupuri de thread-uri şi orientare către prelucrări masive de date.

5.1.Obiective

- Prezentarea conceptelor de programare paralelă CUDA
- Prezentarea unor exemple de algoritmi accelerați cu această bibliotecă
- Utilizarea unor instrumente pentru analiza de performanţă

5.2. Concepte

Plăcile grafice au devenit atât de puternice încât sunt folosite pentru calcul matematic divers, cum ar fi operaţii complexe cu matrici, vectori operaţii necesare pentru diverse simulări vizuale şi fizice complexe în diverse domenii cum ar fi :industria auto, prelucrări video şi de imagini,criptografie,design electronic,simulări fizice diverse, prelucrări multimedia. Unitatile tip GPU sunt potrivite pentru paralelismul de date, intensiv computationale. Datorită faptului că aceleaşi instrucţiuni sunt executate pentru fiecare element, nu sunt necesare mecanisme complexe pentru controlul fluxului. Ierarhia de memorie este simplificată comparativ cu cea a unui procesor x86/ARM. Deoarece calculele sunt intensive computaţional, latenţa accesului la memorie poate fi ascunsă prin paralelism (massive multithreading, SIMT sau Single Instruction Multiple Threads) în locul folosirii extensive a memoriei cache.

NVIDIA a susţinut această tendinţă prin lansarea bibliotecii CUDA (Compute Unified Device Architecture), pentru a permite dezvoltatorilor de aplicaţii să scrie cod care poate fi încărcat pe o placă de tip NVIDIA pentru a fi executat de GPU-urile acesteia .Multe aplicaţii care procesează seturi mari de date, pot utiliza un model de programare bazat pe paralelismul de date pentru a accelera calculele, de exemplu în randarea 3D procesarea de imagini diverse seturi mari de pixeli şi arce sunt mapate către threaduri paralele.

Spre deosebire de CPU , GPU-urile sunt optimizate pentru calcul paralel pe multiple seturi de date, diferenţa majoră între arhitectura CPU şi cea a GPU constă în modalitatea de acces la memorie: CPU-ul foloseşte între 1 şi 3 canale de memorie de câte 64 biţi lăţime fiecare, în timp ce GPU-urile actuale pot folosi până la 8 canale de memorie paralele, de 64 biţi lăţime fiecare. nVidia a identificat acest potenţial al GPU-urilor şi a proiectat o interfaţă de programare prin care se permite anumitor programe accesul la această putere de calcul. Acest model a fost implementat în toate GPU-urile începand cu seria 8800, fiind suportat în principal pe GPU-urile NVIDIA construite pe arhitectura Tesla, iar plăcile grafice care susţin execuţia aplicaţiilor ce folosesc biblioteca CUDA sunt seriile GeForce 8, Quadro, şi Tesla.

Platforma CUDA reprezintă un nivel software care oferă acces direct la setul de instrucţiuni virtuale din GPU şi la componentele software specifice pentru executarea aşa numitelor nuclee de calcul, fiind concepută pentru a lucra cu limbaje de programare precum C, C ++ şi Fortran.Spre deosebire de API-urile pentru procesare grafică cum ar fi Direct3D şi OpenGL, care necesită abilităţi avansate în programarea grafică, această nouă abordare facilitează specialiştilor în programarea

paralelă să utilizeze resursele GPU relativ simplu. Complementar, CUDA susține cadre de programare de nivel mai înalt precum OpenACC și OpenCL. OpenCL, ca standard alternativ suportat de Khronos și implementat de majoritatea producătorilor de GPU (inclusiv NVIDIA ca o extensie la CUDA). Problema majoră însă constă în faptul că suportul oferit este incomplet și standardul este mult mai restrictiv decât CUDA și cu complexitate mai mare în programare.

5.2.1. Modelul de programare

Modelul de programare CUDA este un model heterogen în care este utilizat atât CPU pentru controlul programului, cât și GPU pentru prelucrările masive de date. Placa grafică GPU ,numită în terminologia CUDA dispozitiv (engl.device), reprezintă un co-procesor "multithread" al CPU numit gazdă (engl. host) care poate executa un număr foarte mare de thread-uri identice în paralel. Thread-urile CUDA sunt threaduri "ușoare" (engl. light-threads),ceea ce înseamnă că mecanismele suplimentare necesare pentru crearea acestora, comutarea contextului și planificarea lor sunt foarte rapide. Scopul final al unei procesări în modelul CUDA este să se creeze cât mai multe astfel de threaduri în așa fel încât să se utilizeze cât mai complet și optim resursele hardware disponibile oferite de procesorul grafic. Intrucât procesoarele grafice pot rula mii de thread-uri concomitent, acestea devin eligibile pentru aplicații ce permit un grad mare de paralelizare, astfel principalul avantaj al arhitecturii CUDA este capacitatea de procesare masivă de date într-un interval de timp rezonabil și limitat.

Stiva software CUDA este ilustrată în Figura 5.1 ,iar în modelul de programare se folosesc următoarele concepte :

- *Host/Gazdă* = CPU (unitatea centrală de procesare) + memoria sa
- *Device/Dispozitiv* = set de multiprocesoare GPU (unitatea de procesare grafică) + memoria sa
- *Multiprocesor* = set de procesoare + memorie partajată
- *Bloc* (bloc de thread-uri) = grup de threaduri care execută un cod unic (kernel) pe un set de date identificate de threadID și blockID. Thread-urile pot comunica prin memoria partajată
- *Kernel* = programul (codul) asociat fiecărui thread executat în GPU
- *Grid* = grup (arie) de blocuri de thread-uri care execută kernelul
- *Warp* = grup de thread-uri care pot fi planificate simultan pentru execuție (de ex. dimensiunea unui warp = 32)

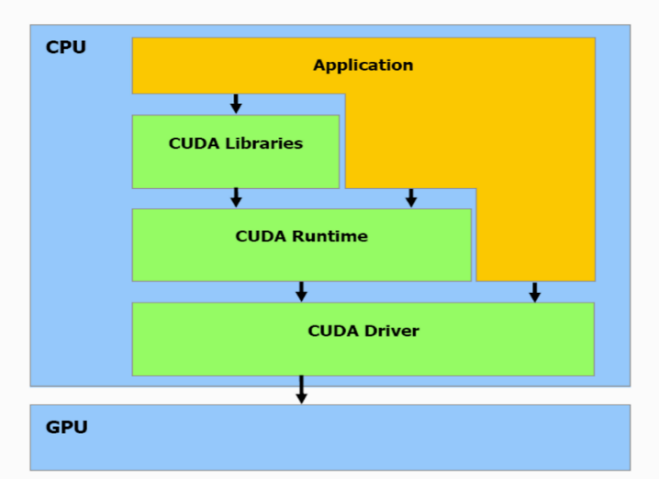


Figura 5.1. Stiva software CUDA [2]

Platforma CUDA utilizează un limbaj similar limbajului C ++ , care posedă extensii pentru a folosi caracteristicile specifice GPU-urilor și anume : apeluri specifice modelului de memorie partajată și pentru sincronizare între thread-uri, respectiv calificatori care se aplică funcțiilor și

variabilelor pentru diferențierea lor. Funcțiile specifice CUDA sunt numite kernel-uri, un astfel de kernel poate fi o funcție sau un program complet invocat de către CPU, ce este executat de N ori în paralel pe un GPU folosind un număr N de thread-uri.

Programarea CUDA implică rularea codului pe două platforme diferite, concomitent: un sistem gazdă cu unul sau mai multe procesoare și unul sau mai multe dispozitive CUDA NVIDIA GPU activate. De cele mai multe ori GPU-urile NVIDIA sunt frecvent asociate cu procesarea grafică, însă acestea sunt de asemenea motoare aritmetice puternice, capabile sa ruleze mii de thread-uri în paralel.Compilatorul NVIDIA (nvcc) este capabil să separe codul sursă în: funcții dispozitiv – procesate de compilatorul NVIDIA, respectiv funcții gazdă – procesate de compilator standard (de exemplu, gcc).

Codul care rulează pe host poate gestiona memoria atât pe aceasta cât și pe dispozitiv. Tot din host se lansează și kernel-urile pentru a fi executate de dispozitiv (placa grafică cu procesoarele sale). Având în vedere caracterul heterogen al modelului de programare CUDA, o secvență tipică de operații pentru un program CUDA C/C++ conține 5 pași ,iar fluxul de procesare este ilustrat în Figura 5.2.:

1. declararea și alocarea memoriei gazdă
2. inițializarea datelor gazdă
3. transferare date de la gazdă la dispozitiv
4. executarea unuia sau mai multor kernel-uri pe dispozitiv în mod paralel
5. transferul rezultatelor de la dispozitiv la gazdă pentru afișare și eliberarea memoriei

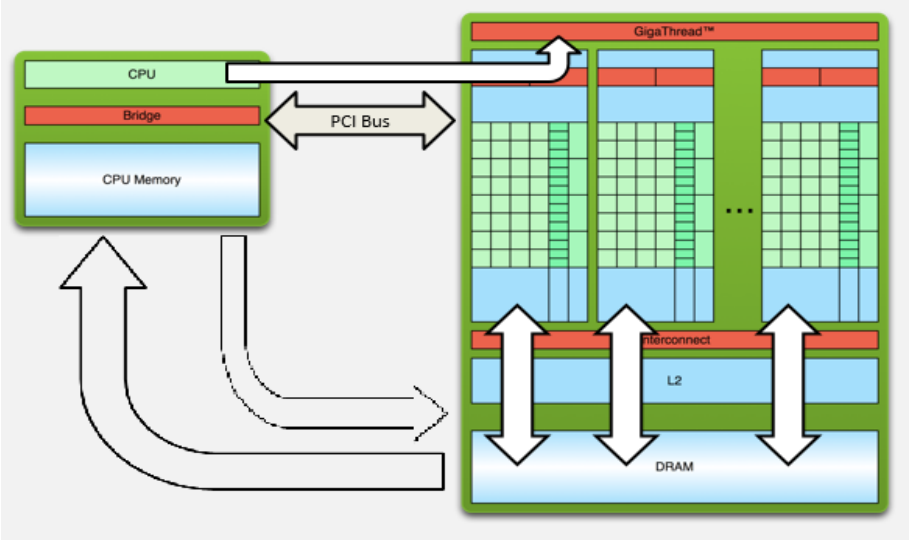


Figura 5.2. Flux de procesare CUDA [2]

Un exemplu de cod scris în CUDA pentru *adunarea a doi întregi*:

__global indică o funcție care: rulează pe dispozitiv și este apelată din gazdă

NUME_Kernel <<< Numar_blocuri_grila, Numar_thread-uri_bloc >>>(parametrii); marchează un apel din codul gazdă la codul dispozitiv (”Kernel launch”)

Exemplu: Adunarea a 2 numere întregi

```
//un kernel simpul pentru adunarea a 2 întregi
__global__ void add(int *a, int *b, int *c)
 { *c = *a + *b; }

int main(void) {
    int a, b, c;    // Copiile variabilelor a, b, c  de pe gazda
```

```
int *d_a, *d_b, *d_c;  // Copiile variabilelor a, b, c  de pe dispozitiv
int size = sizeof(int);
// Alocare spatiului pentru copiile variabilelor a, b, c de pe dispozitiv
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Setarea valorilor de intrare
a = 2;
b = 7;
// Copierea valorilor de intrare în dispozitiv
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
// Lansarea kernelului add() pe GPU
add<<<1,1>>>(d_a, d_b, d_c);
// Copierea rezultatelor inapoi pe gazda
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
// Eliberarea memoriei
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0; }
```

Execuția paralelă este descrisă de funcția kernel care este executată pe un set de thread-uri în paralel pe GPU. Acest cod kernel este un cod C ++ pentru doar un singur thread. Numărul de blocuri thread și numărul de thread-uri din acele blocuri care execută acest kernel în paralel sunt date explicit la apelul funcției. O funcție kernel poate fi invocată în cadrul codului serial, iar pentru a apela funcția kernel *configurația specifică execuției* trebuie specificată și anume numărul de thread-uri într-un bloc thread și numărul de thread-uri dintr-un grid.

Biblioteca CUDA conține următoarele componente : o componentă care rulează pe host și oferă funcții de control și accesare a unuia sau mai multor dispozitive de calcul host, o componentă dispozitiv, care rulează pe dispozitiv și oferă funcții specifice acestuia, o componentă comună, care oferă tipuri încorporate de vectori și un subset de funcții al bibliotecii standard C++, ce sunt acceptate atât în codul gazdă, cât și în cel al dispozitivului. Aceste componente și utilitatea lor sunt prezentate în Tabelul 6.1.

Componenta host	Suportă dispositive multiple
	Memoria: liniară sau tablouri CUDA
	Interoperabilitate OpenGL și DirectX
	Asincronicitate: funcția __global__ și cele mai multe funcții revin în aplicație anterior momentului în care dispozitivul încheie execuția taskului solicitat
Componenta dispozitiv	Funcții de sincronizare
	Conversie de tip
	Casting de tip
	Funcții atocmice (operații de tip read-modify-write pe cuvinte de 32 biti din memoria globală)
Componenta comună	Tipuri Built-in de vectori (float1, float2, int3, ushort4, etc)
	Creare constructor de tip : int2 i = make int2(i, j)
	Funcții matematice (standard math.h pe CPU)
	Funcții de timp pentru benchmarking
	Referințe la structuri de threaduri

Tabel 5.1. Componente CUDA și semnificația lor

Pentru a declara un grid și un bloc thread, CUDA oferă un tip de date predefinit, un vector de tip integer care specifică dimensiunile grid-ului și a blocurilor thread. În apelul funcției kernel variabilele grid și bloc sunt scrise între trei paranteze unghiulare <<< grid, block >>>, în acest mod în apel , grid-ul și blocurile thread sunt create *dinamic.*

Valorile acestor variabile grid și bloc trebuie să fie mai mici decât dimensiunile permise maximale. Funcția kernel are întotdeauna un tip de returnare void și un calificator __global__ care înseamnă că aceasta este o funcție kernel care urmează a fi executată pe GPU.

5.2.2. Managementul threadurilor

Arhitectura CUDA este construită în jurul unei *matrice scalabile de multiprocesoare* cu mai multe fire de execuție (numite și Multithreaded Streaming Multiprocessors - SMS). Când un program CUDA de pe host invocă o grilă kernel, blocurile rețelei sunt enumerate și distribuite multiprocesoarelor ce au capacitate de execuție disponibilă. Thread-urile unui bloc de thread-uri se execută simultan pe un singur multiprocesor, deasemenea mai multe blocuri de thread-uri se pot executa concurent pe un singur multiprocesor.

Pentru a rula cod paralel pe dispozitiv (de exemplu, funcția add() să se execute de N ori în paralel, pentru adunarea a 2 vectori), sintaxa este:

add<<<1,1>>> → add<<<N,1>>>

Fiecare invocare paralelă a lui add() se numește **bloc**.

Un set de blocuri formează un *grid.*

Fiecare invocare a funcției se poate referi la indexul blocului său utilizând *blockIdx.x*

*Exemplu***:**

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];}
```

Kernelurile CPU sunt proiectate pentru a minimiza latența pentru unul sau mai multe thread-uri ce se execută întrețesut utilizând secvențial procesorul, în timp ce GPU-urile sunt proiectate să se ocupe de un număr foarte mare de thread-uri concurente în scopul de a maximiza randamentul în procesare prin creșterea vitezei de execuție.În CUDA, un bloc poate fi împărțit în thread-uri paralele. Referirea la indexul unui thread se face utilizând *threadIdx.x* (vezi Figura 5.3.).

Modelul CUDA structurează un calcul paralel folosind abstracțiunile threaduri, blocuri și griduri. Thread-ul este doar o execuție a unui kernel cu un anumit index. Fiecare thread folosește indexul său pentru a avea acces la elemente, astfel încât thread-urile procesează în mod cooperativ întregul set de date.

Programul are două secțiuni – una serială, care se execută în CPU și una paralelă care se execută în fiecare din procesoarele din GPU, în paralel. In codul sursă va exista o secțiune serială și unul sau mai multe apeluri ale kernel-ului .Organizarea aplicației se va realiza ca în Figura 5.3. în manieră grilă, blocuri și thread-uri, astfel o grilă conține mai multe blocuri și un bloc are mai multe threaduri.

*Exemplu***:** Adunare cu un bloc cu mai multe thread-uri:

```
// Lansarea kernel-ului add() pe GPU cu N blocuri și THREADS_PER_BLOCK thread-uri
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

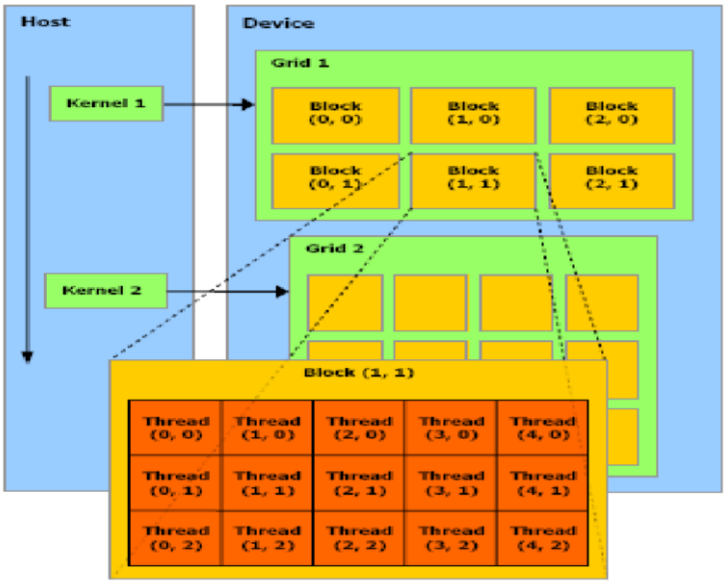


Figura 5.3. Structuri de threaduri

5.2.3. Modelul de memorie şi modelul hardware

Memoria partajată este rapidă în comparație cu memoria device și în mod normal necesită aceeași durată de timp cât este necesară pentru a accesa registrele, fiind numită și Parallel Data Cache (PDC). Memoria partajată este "locală" fiecărui multiprocesor spre deosebire de memoria device și permite astfel o sincronizare locală mult mai eficientă. Aceasta este divizată în mai multe părți, astfel fiecare bloc thread dintr-un multiprocesor își accesează partea *proprie* de memorie partajată și această memorie partajată nu este accesibilă celorlalte blocuri thread ale acelui multiprocesor sau al oricărui alt multiprocesor. Toate thread-urile dintr-un bloc thread care au aceeași durată de viață ca și blocul din care fac parte utilizează această parte a memoriei atât pentru operații de citire cât și de scriere. Întrucât spațiul de memorie partajată este relativ limitat,aceasta trebuie folosită eficient.

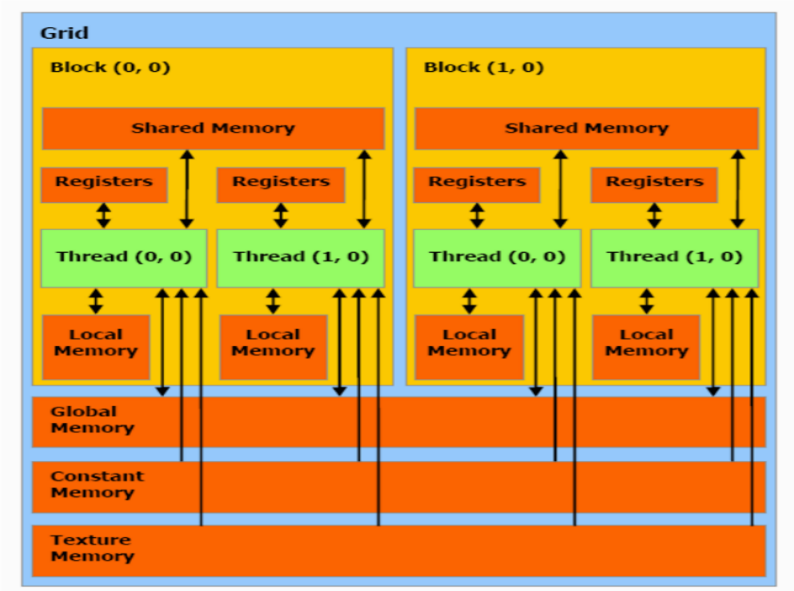


Figura 5.4. Modelul de memorie [2]

Toate multiprocesoarele accesează o memorie mare device globală atât pentru operații gather cât și pentru operații scatter, memorie care este relativ înceată deoarece nu oferă caching.

Fiecare multiprocesor are și cache-urile sale read-only pentru a crește viteza operațiilor de citire. Acestea sunt memoriile constant cache și texture cache.Fiecare thread conține de asemenea memoria lui locală. În mod normal variabilele locale ale funcțiilor kernel sunt alocate aici, însă pot fi alocate și în memoria globală. Într-un bloc, thread-urile partajează datele între ele utilizând *shared memory*.Memoria dispozitivului se numește *memorie globală*. Datele nu sunt vizibile thread-urilor din alte blocuri.

Pentru a declara variabile in memoria partajată se utilizează calificatorul **__shared__** și pentru a declara în memoria globală se folosește calificatorul **__device__.**

Dispozitivul este implementat ca un set de multiprocesoare așa cum este ilustrat în Figura 5.4. Fiecare multiprocesor are o arhitectură de tip SIMD (o singură instrucțiune pe date multiple), astfel la orice ciclu de ceas dat, fiecare procesor al multiprocesorului execută aceeași instrucțiune, dar operează pe date diferite.

Fiecare multiprocesor are o memorie on-chip de unul din următoarele patru tipuri: (i) un set de registre locale pe 32 de biți pe procesor; (ii) un cache de date paralel (memorie partajată) care este partajat de toate procesoarele și implementează un spațiu de memoria partajată (iii) o memorie cache constantă numai pentru citire care este partajată de către toate procesoarele și care accelerează citirile din spațiul de memorie constant și care este implementată ca o regiune de memorie a dispozitivului numai pentru citire; (iv) un cache doar pentru citire care este partajat de toate procesoarele și care accelerează citirile din spațiul de memorie a texturii, care este implementat ca regiune de memorie numai pentru citire din dispozitiv .

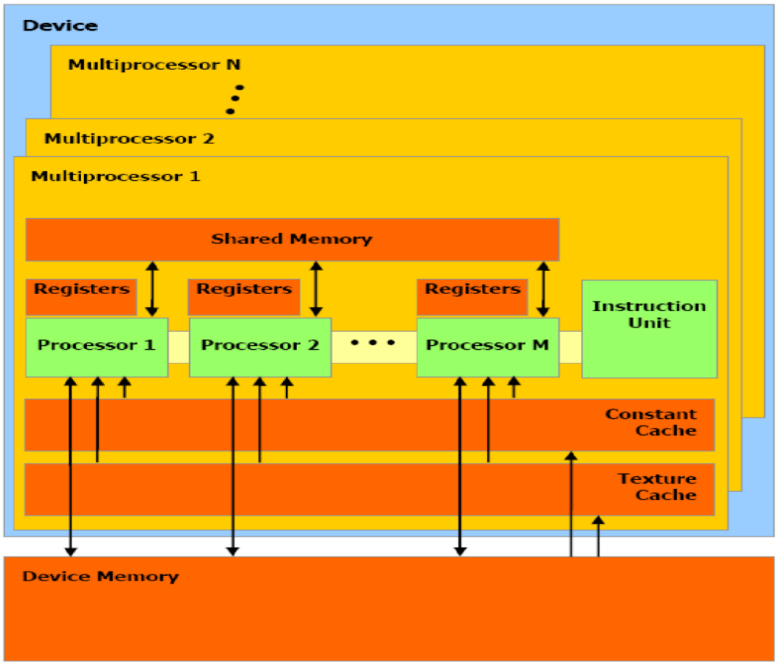


Figura 5.5. Modelul hardware [2]

Spațiile de memorie locale și globale sunt implementate ca regiuni de citire și scriere ale memoriei dispozitivului și nu sunt stocate în cache. Fiecare multiprocesor accesează cache-ul texturii printr-o unitate de textură care implementează diversele moduri de adresare și filtrarea datelor.

O grilă de blocuri de threaduri este executată pe dispozitiv prin executarea unuia sau mai multor blocuri pe fiecare multiprocesor , astfel fiecare bloc este împărțit în grupuri SIMD ale threadurilor numite warpuri; fiecare din aceste warpuri conține același număr de threaduri, numit mărimea warp, și este executat de multiprocesor într-un mod SIMD; un planificator de threaduri trece periodic de la un warp la altul pentru a maximiza utilizarea resurselor computaționale ale multiprocesorului.Un bloc este procesat de un singur multiprocesor, astfel încât spațiul de memorie partajat se află în memoria partajată pe-chip, ceea ce duce la accesări foarte rapide . Regiștrii multiprocesorului sunt alocați între threadurile blocului. Dacă numărul de registre utilizate pe thread

înmulțit cu numărul de threaduri din bloc este mai mare decât numărul total de registre per multiprocesor, blocul nu poate fi executat și nucleul corespunzător nu se va lansa. Mai multe blocuri pot fi procesate simultan de același multiprocesor prin alocarea registrelor multiprocesorului și a memoriei partajate între blocuri

Modul cum accesam memoria impacteaza foarte mult performanța sistemului. Cum putem avea arhitecturi foarte diferite din punct de vedere al ierarhiei de memorie este important de înteles ca nu putem dezvolta un program care să ruleze optim în toate cazurile. Un program CUDA este portabil și astfel poate fi usor rulat pe diferite arhitecturi NVIDIA CUDA, insa de cele mai multe ori trebuie ajustat în funcție de arhitectură pentru o performanță optimă.In general pentru arhitecturile de tip GPU, memoria locală este impărțită în module de SRAM identice, denumite bancuri de memorie (memory banks). Fiecare banc conține o valoare succesiva de 32 biți (de exemplu, un int sau un float), astfel încât accesele consecutive intr‑un array provenite de la threaduri consecutive să fie foarte rapid. Conflicte au loc atunci cand se fac cereri multiple asupra datelor aflate în acelasi banc de memorie.

Conflictele de access la bancuri de memorie (cache) pot reduce semnificativ performanța.Cand are loc un astfel de conflict, hardware‑ul serializează operațiile cu memoria (warp/wavefront serialization), și determină astfel toate threadurile să aștepte până când operațiile de memorie sunt efectuate. In unele cazuri, dacă toate threadurile citesc aceeați adresa de memorie shared, este invocat automat un mecanism de broadcast iar serializarea este evitată. Mecanismul de broadcast este foarte eficient și se recomandă folosirea sa de oricate ori este posibil.

5.2.4. Sincronizarea threadurilor

În scopul de sincronizare a thread-urilor API-ul CUDA oferă o funcție hardware de tip barieră de threaduri (thread-barrier numită *syncthreads()* care se comportă ca un punct de sincronizare. Întrucât thread-urile sunt programate în hardware, această funcție este implementată în hardware. Thread-urile vor aștepta în punctul de sincronizare până când toate thread-urile au ajuns în acest punct. Comunicarea între thread-uri dacă este necesară este posibilă prin memoria partajată a fiecărui bloc. Așadar, sincronizarea thread-urilor este posibilă numai la nivel de bloc thread. Deoarece s-ar putea ca thread-urile unui bloc să comunice între ele, aceste thread-uri trebuie executate pe același processor, de aceea, este garantată execuția unui bloc thread pe un singur processor.

Pentru a maximiza utilizarea resurselor disponibile, asignarea numărului de thread-uri pe bloc și numărului de blocuri thread pe grid ar trebui alese cât mai adecvat. Un număr mai mic de thread-uri pe bloc provoacă latență la încărcare la citirile memoriei device și de asemenea un bloc pe multiprocesor îl poate face pe acesta să fie inactiv în timpul sincronizării thread-urilor. Așadar, ar trebui să fie cel puțin două blocuri pentru fiecare multiprocesor în dispozitiv (numărul de blocuri pe grid ar trebui să fie cel puțin 100). De asemenea, ar trebui să se asigneze numărul de thread-uri pe bloc în multiplii ai dimensiunii warp, deoarece astfel se diminuează numărul warp-urilor subpopulate.

5.2.5. Controlul execuției

Întrucât funcția kernel rulează pe dispozitiv, memoria trebuie alocată în dispozitiv din timp înainte de invocarea funcției kernel și dacă funcția kernel trebuie să execute operații pe anumite date, atunci datele trebuie copiate din memoria host în memoria device anterior procesării.

Memoria device poate fi alocată fie ca memorie liniară, fie ca vectori CUDA. Calificatorul __device__ la începutul unei variabile specifică faptul că spațiul pentru această variabilă este alocat în memoria device. API-ul CUDA are și funcții de alocare și dealocare a memoriei device în timpul execuției, cum ar fi cudaMalloc(), cudaFree() etc. În mod similar, după execuția funcției kernel, datele din memoria device trebuie copiate înapoi în memoria host pentru a putea afișa rezultatele. Pentru a copia date în și din device spre host , API-ul CUDA oferă câteva funcții: *cudaMemCpyToSymbol(), cudaMemCpyFromSymbol(), cudaMemCpy(),* etc.

Lansările de Kernel-uri sunt asincrone, comanda revine imediat la CPU, astfel CPU trebuie să se sincronizeze înainte de a consuma rezultatele.

cudaMemcpy()	Blochează procesorul până când copia este finalizată Copierea începe când au fost terminate toate apelurile CUDA Anterioare
cudaMemcpyAsync()	Asincron, nu blochează CPU-ul
cudaDeviceSynchronize()	Blochează procesorul până când toate apelurile CUDA anterioare au terminat

Apelurile de kernel sunt asincrone din punctul de vedere al procesorului, deci dacă se apeleaza două kerneluri în succesiune, al doilea va fi lansat fără sa mai aştepte terminarea primului.În GPU, în cazul în care nu sunt specificate fluxuri diferite pentru a executa kernel-ul, acestea vor fi executate în ordinea în care au fost apelate ,astfel dacă este specificat un flux, ambele sunt executate în serie şi numai după ce primul kernel este terminat se va executa cel de al doilea .

5.2.6. Tratarea erorilor

Toate funcţiile CUDA au o valoare returnată, care poate fi utilizată pentru a verifica erorile care apar în timpul executării lor (*cudaError).*

- Erori de apeluri API Cuda. De exemplu, un apel la cudaMalloc () ar putea eşua.
- Erori de la apelurile de kernel Cuda. De exemplu, ar putea exista acces la memorie invalidă în interiorul unui nucleu.

Exemplu:

```
if ( cudaSuccess != cudaMalloc( &fooPtr, fooSize ) )
       printf( "Error!\n" );
```

Invocările kernelului CUDA nu returnează nici o valoare. Eroarea de la un apel de kernel CUDA poate fi verificată dupa executarea acestuia prin apelarea metodei cudaGetLastError ():

Exemplu:

```
fooKernel<<< x, y >>>(); // Kernel call
if ( cudaSuccess != cudaGetLastError() )
       printf( "Error!\n" );
```

5.2.7. Gestionarea dispozitivelor

Pentru maşini multi-GPU, utilizatorii pot să selecteze care GPU este ales pentru utilizare. În mod implicit driverul CUDA selecteaza cel mai rapid GPU ca dispozitiv 0. NVIDIA furnizează câteva mijloace prin care pot fi interogate şi gestionate dispozitivele GPU fiind importantă interogarea informaţiilor deoarece, aceasta poate fi folosită în setarea configurării execuţiei kernel-ului la runtime.

- Aplicaţia poate interoga şi selecta GPU-uri:
  - cudaGetDeviceCount(int *count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *device)
  - cudaGetDeviceProperties(cudaDeviceProp *prop, int device)

- Thread-uri multiple pot partaja un dispozitiv
- Un singur thread poate gestiona mai multe dispozitive
  - cudaSetDevice(i) – pentru a selecta dispozitivul curent
  - cudaMemcpy(…)- pentru copii peer-to-peer

5.2.8. Transferul de date între Host și Device

Întrucât lățimea de bandă între memoria device și memoria host este mult mai mică comparativ cu lățimea de bandă între dispozitiv și memoria device care este foarte mare, este de dorit minimizarea transferului de date între host și device. În scop de optimizare este utilă crearea și distrugerea structurilor de date direct în memoria device (în loc să le copiem) și efectuarea de transferuri mari prin combinarea și prelucrarea a mai multor transferuri mai mici pentru a diminua costurile transferului.

5.2.9. Timpul de execuție

Măsurarea timpului de execuție al unui program CUDA se realizează prin definirea unui timer și utilizarea funcțiilor aferente:

```
StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);
sdkStartTimer(&timer);
sdkDeleteTimer(&timer);
```

5.2.10. Calificatori pentru funcții și variabile

Calificatori de Tip de Funcție

Cele trei tipuri principale de calificatori de funcție în CUDA sunt *device, global și host.*

1. *__device__* :Funcțiile cu calificatorul device sunt executate pe dispozitiv. Aceste funcții sunt apelabile numai din device.
2. *__global__ :*Funcțiile cu calificatorul global sunt executate pe dispozitiv dar sunt apelabile numai din host.
3. *__host__* :Funcțiile cu calificatorul host sunt executate pe host și sunt apelabile numai din host.

Când nu se folosește niciun calificator, înseamnă că funcția va rula pe host; este echivalent cu funcțiile declarate cu calificatorul __host__.

Calificatori de Tip de Variabilă

Cele trei tipuri principale de calificatori de variabile în CUDA sunt device, constant, și shared.

1. *__device__* :Variabilele declarate cu __device__ se află pe dispozitiv. Alți calificatori de tip sunt folosiți opționali împreună cu __device__. Dacă o variabilă este declarată numai cu calificatorul __device__ atunci această variabliă se află în memoria globală și are aceeași durata de viață ca aplicația. Întrucât aceasta se află în memoria globală, este accesibilă din toate thread-urile (din grid) și din host prin librăria runtime.

2. *__constant__* :Acest calificator este folosit pentru a aloca constante pe dispozitiv. Este folosit opțional împreună cu calificatorul __device_. Această constantă se află în memoria constant, și are același durata de viață ca aplicația. Este accesibilă din toate thread-urile (din grid) și din host prin biblioteca runtime.

3. *__shared__* :Acest calificator este folosit pentru a aloca variabile shared. Este folosit opțional împreună cu calificatorul __device__. Variabilele shared se află în memoria share a unui bloc thread, și are durata de viață a blocului respectiv. Este accesibilă numai din thread-urile din blocul respectiv.

Variabile Incorporate. Aceasta este o listă cu câteva dintre variabilele încorporate în CUDA:

1. gridDim: este de tip dim3 și conține dimensiunile grid-ului.
2. blockIdx: este de tip uint3 și conține indexul blocului din grid.
3. blockDim: este de tip dim3 și conține dimensiunile blocului.
4. threadIdx: este de tip uint3 și conține indexul thread-ului din bloc.
5. warpSize: este de tip int și conține dimensiunea warp-ului în thread-uri.

Componenta Device Runtime. Componentele device runtime sunt folosite numai în funcțiile device și sunt prefixate o liniuță de subliniere __. Cele mai importante funcții sunt :funcții Matematice, de Sincronizare, Atomice, de Texturare.

Modul Device Emulation. Un mod device emulation este oferit în scopuri de debugging. Opțiunea –device nu este folosită împreună cu comanda nvcc compile. Aceasta emulează doar dispozitivul. Thread-urile și blocurile thread sunt create în host.

Debugging-ul nativ al host-ului (ca de exemplu Visual Studio de la Microsoft) poate fi folosit pentru a seta breakpoints și pentru inspectarea datelor, fiind în special folositor în operațiile de input sau output spre fișiere sau spre ecran, cum ar fi utilizarea funcției printf(), care nu se poate rula pe dispozitiv.

5.2.12. Reguli și elemente de optimizare în scrierea codului

Un set minim de extensii la limbajul C, care permit programatorului să vizeze porțiuni ale codului sursă pentru a fi executate pe dispozitiv Cele mai importante aspecte ce vizează regulile necesare în implementarea unor aplicații care să beneficieze optim de modelul CUDA sunt :

1. Programarea simplă C este suportată de către compilatorul CUDA. Lipsesc folosirea funcțiilor de programare orientată pe obiect sau a funcțiilor C++ în codul dispozitivului.
2. Arhitectura eterogenă este folosită pentru a realiza interacțiunea între modelele de programare CPU și GPU. Datele ar putea să fie copiate din memoria host în memoria device, iar rezultatele sunt copiate înapoi în host din memoria device.
3. Invocarea funcției kernel: Grid-ul, blocurile thread, și thread-urile sunt create de către invocarea funcției kernel din host. Acesta este singurul mod de a le crea. Acestea nu pot fi create în funcția kernel. Mai mult, numărul de grid-uri și de blocuri thread nu trebuie să depășească valorile lor maxime permise.
4. Funcțiile kernel nu returnează niciun rezultat, adică tipul lor de returnare este mereu void. Mai mult, apelul funcției kernel este asincron. Așadar, controlul revine înapoi înainte de completarea funcției kernel pe dispozitiv. Mai multe informații pot fi găsite în ghidul de programare CUDA. Toate funcțiile cu calificativul __device__ sunt implicit inline.
5. Recursivitatea este pur și simplu interzisă în funcțiile kernel din cauza cerințelor mari de memorie pentru mii de thread-uri.
6. Alocarea și dealocarea memoriei device în timpul rulării este posibilă numai în timpul utilizării codului host și înaintea apelării codului device. Asta înseamnă că în codul device, memoria device nu poate fi nici alocată nici dealocată folosind funcții precum cudaMalloc(), cudaFree() etc. Toate alocările necesare pentru o funcție kernel specifică sunt realizate înaintea apelului acelei funcției kernel din codul host și, similar, toată memoria device alocată este dealocată după finalizarea acelei funcții kernel din codul host.
7. Memoria partajată este împărțită numai între thread-urile din același bloc thread. Thread-urile dintr-un bloc thread diferit nu o pot împărți.
8. Variabilelor incorporate cum ar fi blockIdx, threadIdx, etc, nu le pot fi asignate nicio valoare. Mai mult, nu este posibilă preluarea adreselor acestora.

9. Variabilele declarate cu calificatorii __device__, __shared__, sau __constant__ au și câteva restricții. Adresa unei variabile folosind oricare dintre acești calificatori poate fi folosită numai în codul device.

Comunicarea și sincronizarea între thread-uri sunt posibile numai la nivelul de bloc thread., iar comunicarea între blocuri thread nu este permisă. O posibilă problemă ar fi concurența și consistența datelor. Atunci când mai multe thread-uri accesează aceeași zonă de memorie, trebuie asigurat faptul că datele sunt consistente. Se pot folosi urmatoarele soluții:

- operații atomice (in care un singur thread are acces la o anumită resursă la un anumit moment de timp)
- sincronizarea threadurilor
- proiectarea sau modificarea algoritmului ăn asa maniera încât să nu apară conflicte

O altă problemă o constituie accesul la memorie. Intrucât accesele la memorie sunt mult mai lente decât timpul de realizare a diverselor instrucțiuni, apare situația în care procesorul grafic trebuie să aștepte (stă neutilizat-idle) până o anumită operație cu memoria este îndeplinită. Pentru acest lucru se pot folosi optimizări precum:

- utilizarea memoriei mai rapide: partajată și locală
- accesul uniform la memorie (coalesced access): evitarea salturilor în memorie pentru date conectate logic și gruparea lor pentru eficientizarea acceselor
- evitarea ramificării codului: evitarea instrucțiunilor condiționale în care ramurile să aibă timpi de prelucrare disproportionați, deoarece thread-urile se lansează în grupuri (warp-uri) de câte 32, iar fiecare thread așteaptă după terminarea execuției celorlalte thread-uri
- paralelizarea apelurilor: apelurile pentru GPU de manipulare a memoriei și execuție de funcții pot fi făcute pe canale (stream-uri) distincte, astfel încât execuția uneia să fie condiționată de terminarea execuției tuturor celorlalte

NVIDIA Visual Profiler este o aplicatie, parte a NVIDIA CUDA Toolkit, care poate să ofere multe informatii cu privire la problemele de performanță dintr‑o aplicație CUDA. Se crează o sesiune si se indică binarul împreună cu argumente la care se dorește profilare.Sunt oferite informații detaliate asupra timpilor de execuție, ocuparea resurselor GPU cât și indicații despre cum se poate îmbunătăți performanța (informații suplimentare : https://docs.nvidia.com/cuda/profiler‑usersguide/index.html)

5.3. Exemple

5.3.1.Exemplu de structură de cod CUDA (sursa [3])

Program	Observatii
const int N = 1024; const int blocksize = 16;	Set grid size

__global__
void add_matrix(float* a, float *b, float *c, int N)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y +
threadIdx.y; int index = i + j*N; if (i < N
&& j < N)
c[index] = a[index] +
b[index]; }
	Compute kernel

int main() {	

float *a = new float[N*N];
float *b = new float[N*N];
float *c = new float[N*N];
for (int i = 0; i < N*N; ++i) a[i] = 1.0f; b[i] = 3.5f;
	CPU memory allocation

float *ad, *bd, *cd;
const int size = N*N*sizeof(float);
cudaMalloc((void**)&ad, size);
cudaMalloc((void**)&bd, size);
cudaMalloc((void**)&cd, size);
	GPU memory allocation

cudaMemcpy(ad, a, size, cudaMemcpyHostToDevice); cudaMemcpy(bd, b, size, cudaMemcpyHostToDevice);	Copy data to GPU

dim3 dimBlock(blocksize, blocksize);
dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
add_matrix<<<dimGrid, dimBlock>>>(d, bd, cd, N);
	Execute kernel

cudaMemcpy(c, cd, size, cudaMemcpyDeviceToHost);	Copy result back to CPU
cudaFree(ad); cudaFree(bd); cudaFree(cd); delete[] a; delete[] b; delete[] c; return EXIT_SUCCESS;	Clean up and return
}	

Instrucțiuni de execuție a exemplului de aplicație. Pentru a folosi modelul de programare masiv paralelă CUDA în implementarea de aplicații este nevoie de:

- Un GPU care suportă CUDA
- O versiune compatibilă de Windows
- O versiune compatibilă de Microsoft Visual Studio
- NVIDIA CUDA Toolkit (disponibilă la http://developer.nvidia.com/cuda-downloads) –versiunea curentă este 9.1

Pentru crearea unui nou proiect se vor urma etapele :New Project -> NVIDIA -> CUDA 9.1 -> CUDA 9.1 Runtime.
Pentru configurare se vor alege proprietățile corespinzătoare , astfel : -> Properties -> Configuration Properties -> VC++ Directories , unde se vor realiza următoarele configurări:
Include Directories: C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.1\common\inc\
Library Directories: C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.1\common\lib\

Pentru implementarea paralelă utilizând frameworkul CUDA se poate utiliza librăria Cuda C++ Thrust (https://developer.nvidia.com/thrust) care permite implementarea aplicațiilor paralele de înaltă performanță cu un efort mai mic de programare. Performanța execuției poate fi măsurată utilizând deasemenea extensia Concurency Visualizer din Visual Studio 201x.

5.3.2.Exemplu de paralelizare a unui program secvențial care calculează distanța între un punct specific și toate celelalte puncte dintr-o matrice 2D de dimensiune NxN (sursa [4]).

Cod secvențial

```
const int N=16;
void main (void) {
    int i, j, x, y;
    float hgrid[N][N];
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &x);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &y);
    // Code to find distance without using device
```

```
    for (i=0; i<N; i++){
        for (j=0; j<N; j++) {
            n = ((i-x)*(i-x))+((j-y)*(j-y));    // distance formula
            hgrid[i][j] = sqrt(n);              // distance formula
            printf("\t%.0lf", hgrid[i][j]);}
        printf("\n\n");}}
```

Cod Paralel - 1D Grid

Același program este convertit în cod paralel pentru a fi rulat pe dispozitiv.
Un grid unidimensional cu numai un singur bloc thread este folosit. Blocul thread conține 16*16 thread-uri (așadar 256 de thread-uri în total) într-o formă bidimensională.

```
const int N=16;
__device__ float dgrid[N][N]; // array on device memory

// function on device to calculate distance
__global__ void findDistance( int x, int y){
    int i = threadIdx.x;
    int j = threadIdx.y;

    float n = ((i-x)*(i-x))+((j-y)*(j-y));
    dgrid[i][j] = sqrt(n);}

void main () {
    int i, j;
    float hgrid[N][N];
    dim3 dBlock(N, N);      // thread block with total 256 threads
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
    printf( "\n\tDistance from a node!\n\n\n" );

    findDistance<<<1, dBlock>>>(i, j); // Calling kernel function
    cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid)); //copy device M to host

    printf( "\n\n\tValues in hgrid!\n\n" );
    for (i=0; i<N; i++){
        for (j=0; j<N; j++) {
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n"); }}}
```

*Cod Paralel - 2D Grid (2 * 2)*

Același program este convertit în cod paralel pentru a fi rulat pe dispozitiv cu un grid bidimensional (2 blocuri thread în dimensiunea x și 2 în dimensiunea y). Blocul thread conține 16 * 16 thread-uri (adică 256 de thread-uri în total) într-o formă bidimensională. Așadar, în total vor rula 1024 de thread-uri în paralel pe dispozitiv.

```
const int N=16;
const int D=2;
__device__ float dgrid[N*D][N*D]; // array on device memory

// function on device to calculate distance
__global__ void findDistance( int x, int y){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    float n = ((i-x)*(i-x))+((j-y)*(j-y));
    dgrid[i][j] = sqrt(n);}

void main () {
    int i, j;
    float hgrid[N*D][N*D];
    dim3 dGrid(D,D);    // 2D grid with total 4 thread blocks
    dim3 dBlock(N, N);  // thread block with total 256 threads
```

```
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
    printf( "\n\tDistance from a node!\n\n\n" );

    findDistance<<< dGrid, dBlock>>>(i, j); // Calling kernel function
    cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid)); //copy device M to host

    printf( "\n\n\tValues in hgrid!\n\n" );
    for (i=0; i<N*D; i++){
        for (j=0; j<N*D; j++){
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n");}}}
```

Cod Paralel - 2D Grid (4 * 4)

Același program este convertit în cod paralel pentru a fi rulat pe dispozitiv cu un grid bidimensional (4 blocuri thread în dimensiunea x și 4 în dimensiunea y). Blocul thread conține 8 * 8 thread-uri (adică 64 de thread-uri în total) într-o formă bidimensională. Așadar, în total vor rula 1024 de thread-uri în paralel pe dispozitiv.

```
const int N=8;
const int D=4;
__device__ float dgrid[N*D][N*D]; // array on device memory

// function on device to calculate distance
__global__ void findDistance( int x, int y){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    float n = ((i-x)*(i-x))+((j-y)*(j-y));
    dgrid[i][j] = sqrt(n);
}

void main () {
    int i, j;
    float hgrid[N*D][N*D];
    dim3 dGrid(D,D);    // 2D grid with total 16 thread blocks
    dim3 dBlock(N, N);  // thread block with total 64 threads
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
    printf( "\n\tDistance from a node!\n\n\n" );
    findDistance<<< dGrid, dBlock>>>(i, j); // Calling kernel function
    cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid));//copy device memory to
host

    printf( "\n\n\tValues in hgrid!\n\n" );
    for (i=0; i<N*D; i++){
        for (j=0; j<N*D; j++){
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n"); }}}
```

5.3.3.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului Cuda, exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/05_CUDA

5.4. Întrebări teoretice

5.4.1.Care este avantajul major al arhitecturii CUDA?
5.4.2.Cum pot fi identificate threadurile create și executate la nivel de device
5.4.3.Prezentați ierarhia de memorie a platformei CUDA
5.4.4.Ce tipuri de calificatori pentru funcții și variabile sunt utilizați în programarea CUDA?

5.5. Probleme propuse

5.5.1.Să se implementeze algoritmii de sortare *radix,merge și bitonic* în manieră paralelă folosind mecanismele modelului de programare CUDA. Să se realizeze o comparație a timpului de execuție a acestora pentru o evaluare de performanță a lor ,raportat la complexitatea algoritmului și dimensiunea problemei, respectiv gradul de paralelism (DOP- degree of parallelism reprezintă număr de threaduri/procese ce s epot executa în paralel).

5.5.2. Implementați utilizând CUDA functia matrix_multiply_simple care va realiza inmultirea a două matrice primite ca parametru. Realizați o inmulțire optimizată a două matrice, folosind metoda Blocked Matrix Multiplication (https://en.wikipedia.org/wiki/Block_matrix). Se va folosi directiva shared pentru a aloca memorie partajata intre thread‑uri. Pentru sincronizarea thread‑urilor se folosește funcția __syncthreads. Măsurați timpul petrecut în kernel pentru fiecare din soluțiile implementate folosind evenimente CUDA. Realizați profilare pentru funcțiile implementate folosind tool‑urile nvprof si nvvp, respectiv NVIDIA Visual Profiler, la alegere.

5.6. Referințe bibliografice

1. Toolkit și tutorial http://developer.nvidia.com/cuda
2. Instrucțiuni pentru instalare și configurare
   http://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html
3. Tutoriale https://www.tutorialspoint.com/cuda/cuda_tutorial.pdf
   https://computing.llnl.gov/tutorials/linux_clusters/gpu/NVIDIA.Introduction_to_CUDA_C.1.pdf
4. Exemple de aplicații : https://github.com/zchee/cuda-sample
5. Librăria Cuda Thrust https://thrust.github.io/

6. Programare distribuită cu socketuri

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

6.1. Obiective

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

6.2. Concepte

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

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

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

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

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

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

6.2.1. Elemente în programarea cu socketuri

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

Metodele statice ale clasei sunt:

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

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

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

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

Constructorii clasei sunt:

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

Un obiect de tip URL poate fi folosit pentru:

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

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

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

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

6.2.2.Socketuri TCP

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

Socket-urile sunt de două tipuri:

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

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

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

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

Constructorii cei mai utilizaţi ai acestei clase sunt:

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

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

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

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

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

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

Constructorii clasei pentru crearea de noi socketuri sunt:

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

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

Metodele clasei

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

Inchiderea unui socket

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

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

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

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

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

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

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

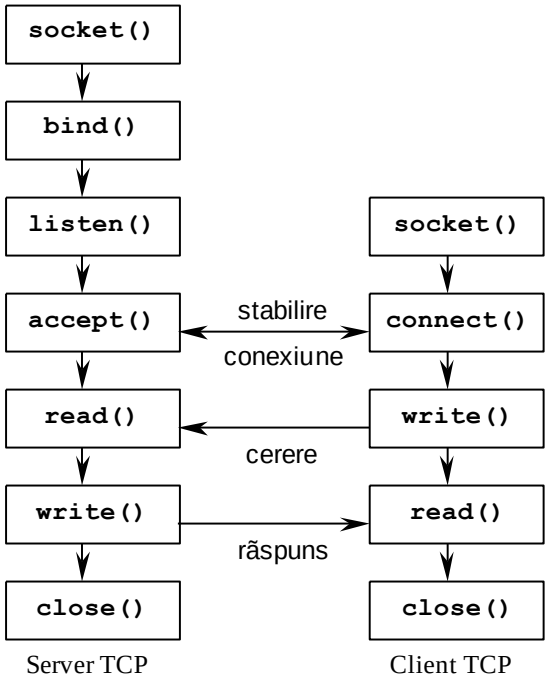


Figura 6.1. Mecanismul socket TCP

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

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

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

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

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

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

Structura generală a unui server bazat pe conexiuni este următoarea:

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

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

Structura generală a unui client bazat pe conexiuni este:

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

6.2.3. Socketuri datagrame

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

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

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

Clasa DatagramPacket

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

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

Metodele acestei clase sunt:

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

Clasa DatagramSocket

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

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

Transmisia şi recepţia datagramelor

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

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

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

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

Acestea sunt: *getAdress, getPort si getSocketAddress.*

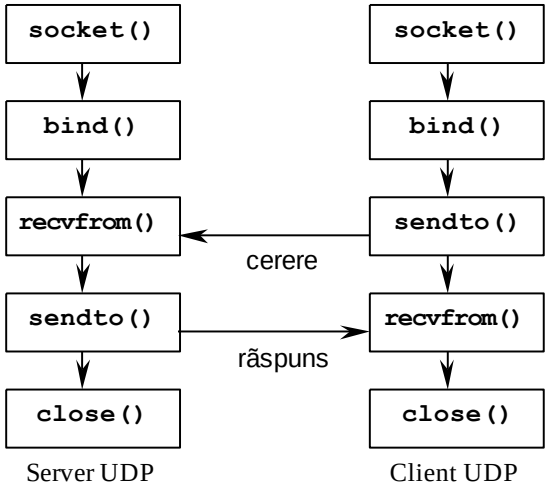


Figura 6.2. Mecanismul socket datagrame

6.2.4.Socketuri multicast

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

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

Ceea ce diferenţiază totuşi comunicarea multicast de cea unicast o constituie specificarea şi gestionarea câmpului TTL (Time to live) câmp ce permite evitarea buclelor de rutare pentru topologii arborescente complexe.Adresele de tip multicast sunt adrese de clasă D în gama de adrese 224.0.0.0 – 239.255.255.255, domeniu structurat pe diferite tipuri de adrese multicast rezervate sau nu.

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

Metodele clasei necesare comunicării cu un grup multicast sunt:

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

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

6.3.Exemple

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

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

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

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

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

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

Serverul

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

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

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

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

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

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

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

6.4.Întrebări teoretice

6.4.1. Care este diferența dintre un socket și un port? Ce reprezintă protocolul

6.4.2.Ce tipuri de socketuri sunt implementate în pachetul Java.net? identificați cele mai importante două elemente distinctive.

6.4.3.Ce aplicabilitate pot avea socketurile multicast? Descrieți un miniscenariu de aplicație.

6.5.Probleme propuse

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

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

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

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

6.6.Referințe bibliografice

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

7.RMI -programare distribuită prin invocarea metodelor obiectelor remote

Mecanismul RMI (engl.Remote Method Invocation) furnizează o modalitate prin care este posibilă execuţia unei aplicaţii distribuite pe mai multe maşini virtuale Java, permiţând ca un set de calculatoare să poată colabora prin transfer de date şi cod pentru rezolvarea unui task comun. Tehnologia RMI furnizează o platformă solidă pentru aplicaţii bazate pe obiect distribuite, fiind utilizabilă atât pentru a conecta obiecte Java între ele, cât şi cu obiecte implementate în alte limbaje, oferind posibilitatea extinderii sistemului distribuit într-un mod incremental, prin adăugarea de noi clienţi și/sau servere Java conform cerințelor .

7.1.Obiective

- Studiul conceptelor şi mecanismelor RMI pentru dezvoltarea unei aplicații distribuite prin apelul la distanță a metodelor unor obiecte aflate pe server , într-o mașină virtuală Java diferită, funcționalitate de care clientul este informat prin specificația în cadrul unei interfețe ce conține semnăturile metodelor ce pot fi invocate
- Cunoașterea capabilităților oferite de serviciul de nume rmiregistry pentru identificarea obiectelor la distanță într-un mediu distribuit
- Studiul claselor și interfețelor din pachetul Java.rmi
- Implementarea unor aplicații distribuite folosind mecanismele oferite

7.2. Concepte

Paradigma RMI a preluat o serie de concepte şi mecanisme din modelul clasic al apelului de proceduri la distanță RPC (engl.Remote Procedure Call) adaptate pentru sisteme obiectuale, astfel ,un obiect "client" poate să apeleze o metodă a unui obiect "server" aflat în alt spațiu de adrese, concept extins cu alte facilităţi specifice programării distribuite obiectuale, cele mai importante fiind:un micro-sistem de numire a entităţilor, colectorul de deşeuri, încărcarea dinamică a claselor, oferind astfel o modalitate elegantă de soluționare a diverselor probleme specifice programării distribuite, pentru medii Java.

7.2.1. Considerații generale

În aplicaţiile distribuite atât datele cât şi codul (executabil pe procesoare multiple în sistem), respectiv utilizatorii care comunică şi interacţionează pot fi distribuiţi. RMI este un sistem de programare obiectual, distribuit construit în platforma Java, facilitate ce permite interacţiuni între obiecte care rulează pe maşini virtuale Java (JVM) distribuite în reţea. Spre deosebire de programarea bazată pe socketuri, mecanismul RMI oferă un context de programare distribuită flexibil şi relativ simplu, reprezentând mecanismul suport pentru diverse alte modele de dezvoltare a aplicaţiilor distribuite Java (de e. Java EE Beans).

RMI permite crearea unei referinţe la un obiect care aparţine unui proces de pe un alt calculator şi invocarea de metode ale obiectului ca şi cum acesta ar fi local. În loc să fie necesară definirea unui protocol de transfer de mesaje şi un format al datelor transmise între procesele aplicaţiei distribuite, se foloseşte *interfaţa Java ca protocol*, iar *argumentele metodei exportate devin formatul datelor transmise.*

Sistemul de programare a aplicaţiilor distribuite RMI oferă dezvoltatorului de aplicaţii următoarele facilităţi:

- *localizarea obiectelor la distanţă*- aplicaţiile folosesc diverse mecanisme pentru a obţine referinţele obiectelor remote (serviciul de nume rmiregistry sau transferul referinţelor la obiecte ca parte a operaţiei solicitate)
- *comunicare cu obiectele distribuite* - detaliile comunicării sunt gestionate de RMI, pentru programator comunicaţia remote este similara unei invocări de metode locale
- *încărcarea codului clasei pentru obiecte transferate ca parametri sau valori returnate*. Deoarece permite ca apelantul să transfere obiecte, RMI integrează mecanismele necesare încărcării codului obiectului şi transmiterea datelor.

Serverul apelează un sistem de localizare a obiectelor (numit rmiregistry) pentru a asocia un nume cu un obiect remote, astfel clientul va căuta obiectul după nume în registry şi apoi va invoca metoda. Pot fi utilizate servere Web pentru a încărca codul intermediar al claselor Java client-server pentru obiectele aplicaţiei ,dacă este necesar folosind orice protocol de tip URL (Uniform Resource Locator) suportat de platforma Java. Cele mai importante avantaje oferite de mecanismul RMI sunt:

- *este orientat obiect*: poate transfera obiecte ca parametri de apel a metodelor remote, dar şi ca valori returnate.
- oferă *cod portabil*: raportat la JVM
- *integrează un* Garbage Collector *distribuit* oferind astfel suport pentru gestionarea referinţelor la obiectele distribuite
- integrează *mecanisme de securitate*: foloseşte controlerul de securitate integrat JVM pentru a proteja sistemele de cod destructiv
- oferă suport pentru procesare paraleă şi comportament mobil:poate să schimbe 'comportamentul', adică implementările claselor, de la client la server şi invers

7.2.2. Arhitectura RMI

Arhitectura sistemului RMI integrează 3 niveluri : nivelul stub/skeleton, nivelul de referinţe la distanţă şi nivelul de transport, fiecare nivel software fiind delimitat de o interfaţă şi un protocol specifice, fiind independent de celelalte şi posibil înlocuibil cu o implementare alternativă (vezi fig. 7.1). Pentru interacţiunile la distanţă sunt folosite două tehnici astfel: pentru a realiza transmiterea transparentă a unui obiect între spaţii de adrese diferite, e folosită *tehnica de serializare a obiectului,* iar pentru a permite clientului apelul metodelor la distanţă e utilizată *tehnica de încărcare dinamică a unui intermediar de acces (stub)* ce implementează acelaşi set de interfeţe la distanţă ca şi obiectul.

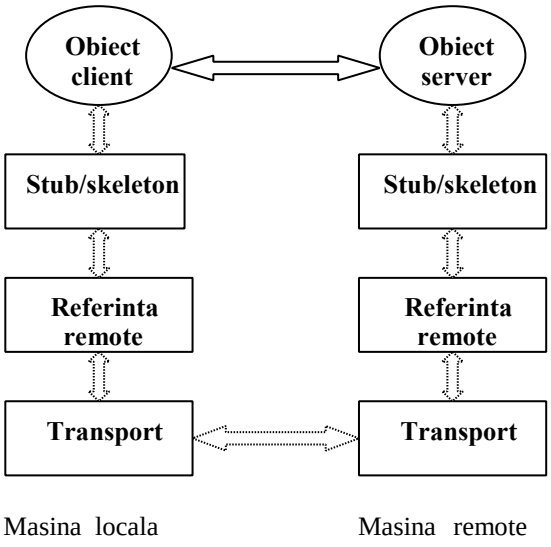


Figura 7.1.Arhitectura RMI şi relaţia client-server

Nivelul Stub/Skeleton - nivelul stub este la client, iar skeleton la server, comportându-se asemenea unor proxy. JVM permite doar invocarea de metode locale, astfel o cerere de invocare a unui obiect de la distanţă parcurge nivelurile sistemului RMI în ambele sensuri, presupunând folosirea unui stub pentru obiectul de la distanţă şi a unei căi spre acel obiect. Acest nivel defineşte interfaţa dintre aplicaţie şi sistemul RMI, nu se ocupă cu detalii de transport, dar transmite datele spre stratul de referinţă la distanţă via abstractizarea numită *stream-uri,* bazat pe mecanismul de serializare, ce susţine transmisia obiectelor Java între diferite spaţii de adrese. Clasele de tip stub -skeleton sunt generate la runtime şi sunt încărcate dinamic.

S*tubul* pentru un obiect remote este proxy-ul la client al acelui obiect, el implementează toate interfeţele care sunt suportate de obiectul remote, fiind reponsabil pentru:

- iniţierea unui apel spre obiectul remote (prin apelul stratului de referinţă la distanţă)
- structurarea argumentelor (argumente sunt obţinute de la stratul de referinţă la distanţă)
- informarea stratului de referinţă la distanţă de faptul că apelul este necesar
- reconversia valorii de return sau a excepţiei dintr-un stream
- informarea stratului de referinţă la distanţă de faptul că apelul a fost efectuat

S*keletonul* pentru un obiect remote este o entitate în server care conţine metode ce efectuează apelurile spre implementarea obiectului remote. Această entitate este responsabilă cu:

- reconversia argumentelor primite
- efectuarea apelului spre implementarea obiectului remote
- returnare sau a excepţiei în stream-ul marshall

Nivelul de referinţă la distanţă integrează comportamentul referinţei la distanţă a obiectului, se ocupă cu interfaţa de nivel jos a nivelului de transport şi cu gestionarea unui protocol specific de referinţă la distanţă independent de stub-rile clienţilor şi skeleton-ii serverelor. Nivelul de referinţa la distanţă are două componente care cooperează astfel: componenta client conţine informaţii specifice despre server-ul remote şi comunică via strat transport cu componenta server. Fiecare implementare de obiect remote îşi alege propria sa subclasă de referinţă care va opera în locul ei, diferite protocoale de invocare pot fi utilizate la nivelul acestuia , ca de exemplu:

- invocare punct la punct unicast
- invocare a grupurilor de obiecte replicate
- suport pentru o strategie specifică de replicare a obiectelor
- suport pentru o referinţă persistentă spre obiectul remote
- strategii de reconectare

Nivelul de transport se ocupă de stabilirea conexiunii şi managementul acesteia, precum şi de identificarea obiectului la distanţă. În general nivelul de transport este responsabil pentru:

- realizarea conexiunii spre spaţiul de adrese îndepărtat
- managementul conexiunilor, monitorizarea stării conexiunii
- aşteptarea de apeluri şi realizarea unei conexiuni pentru un apel cerut de client
- păstrarea unei tabele de obiecte remote care exista în spaţiul de adrese
- localizarea dispecerului pentru destinaţia unui apel la distanţă şi transmiterea conexiunii
-

Reprezentarea concretă a referinţei unui obiect remote constă dintr-un capăt de conexiune şi un identificator de obiect, reprezentare numită "referinţa vie" (live reference). Fiind dată o asemenea reprezentare pentru un obiect remote, un transportor poate folosi capătul de conexiune pentru realizarea unei conexiuni la spaţiul de adrese în care se găseşte obiectul remote. Transportorul conţine 4 abstracţii de bază:

1. *capăt de conexiune* - abstractizarea unui spaţiu de adrese sau a unei JVM, pentru un capăt de conexiune se poate obţine o instanţă specifică pentru transport
2. *canal* - abstractizarea pentru calea dintre două spaţii de adrese, responsabil pentru managementul conexiunilor între spaţii de adrese locale şi spaţiul de adrese remote pentru care este definit canalul
3. *conexiune* - abstractizare pentru transferul datelor

4. *abstracţia de transport* controlează canalele. Fiecare canal e o conexiune virtuală între 2 spaţii de adrese, într-un transport poate exista un singur canal pentru fiecare pereche de spaţii de adrese, abstracţia de transport fiind responsabilă pentru acceptarea apelurilor de pe conexiunile care se stabilesc spre spaţiul de adrese şi pentru pasarea acestui apel spre straturile superioare din sistem pentru acelaşi transport pot exista mai multe implementări (TCP-UDP pentru aceeaşi maşina virtuală).

7.2.3. Mecanisme RMI

Modelul obiectelor distribuite. Numim obiect remote un obiect care oferă metode apelabile de la distanţă. În terminologia client/server, obiectul remote îl vom numi server, iar obiectul care invocă o astfel de metodă îl vom numi client. Localizarea obiectelor la distanţă este realizată prin intermediul unui *server de nume (rmiregister)*, care stochează *referinţe tip URL către obiecte la distanţă* (protocolul din URL poartă numele rmi). Un astfel de serviciu de nume necesită următorul set de funcţiuni: introducere de asocieri nume – obiect, căutări de asocieri, modificări / ştergeri de asocieri, listarea asocierilor existente la un moment dat.

Serializare. Mecanismul prin care un obiect este transformat într-un flux de octeţi şi mai apoi reconstituit, se numeşte serializare, fiind deosebit de util pentru RMI şi alte aplicaţii în reţea.

Pentru transmiterea efectivă la distanţă a parametrilor şi a rezultatelor, RMI foloseşte *mecanismul de serializare* (Object Serialization Protocol), protocol ce se bazează pe mecanismul obişnuit de serializare.Stub-ul şi skeleton-ul se ocupă cu detaliile de reţea (împachetarea şi despachetarea parametrilor şi a valorilor returnate de către metode apelabile la distanţă) transformând parametri şi valorile returnate în fluxuri de octeţi ce pot fi transmise prin reţea.

În Java, orice obiect care implementează interfaţa *java. io. Serializabile* poate fi transformat într-un flux de octeţi şi mai apoi reconstituit din acesta. Această interfaţă nu adaugă nici o metodă necesar a fi implementată, ea este o interfaţă ce poate fi utilizată pentru a permite serializarea claselor, în acest mod pot fi transferate obiecte complexe Java în reţea, fie ca argumente ale unei metode, fie ca valori returnate.

Orice tip pe care îl folosim ca parametru sau valoare returnată la apelul unei metode la distanţă trebuie să fie serializabil. Cerinţe de serializare a unei clase sunt:

- să posede un constructor public fără parametri, necesar pentru deserializarea corectă a obiectelor
- să nu conţină referinţe la obiecte ce nu sunt serializabile (Java nu poate determina complet dacă un obiect este sau nu serializabil în timpul compilării astfel dacă o dată membră a unui obiect nu poate fi serializată, atunci obiectul în sine nu poate fi serializat).

Interfaţa la distanţă. Legătura între server şi client este realizată prin intermediul unei interfeţe de acces la distanţă. Definirea unei astfel de interfeţe trebuie să verifice următoarele condiţii:

- trebuie declarată public
- trebuie să extindă interfaţa java. rmi. Remote
- fiecare metodă din interfaţa la distanţă trebuie să declare posibilitatea de a arunca excepţia java. rmi. RemoteException

Implementarea interfeţei la distanţă respectă *următoarele etape:*

1. obiectul server extinde *java. rmi. server. UnicastRemoteObject şi* implementează interfaţa la distanţă
2. se specifică constructorul obiectului server (si excepţia java. rmi. RemoteException)
3. pentru obiectul server trebuie să se definească şi să se instaleze un manager de securitate (de ex. RMISecurityManager).
4. constructorul obiectului server foloseşte, printre altele, un fişier în care să se specifice "politica de securitate" adoptată de către server (identificarea numelui acestui fişier este realizată prin stringul atribuit variabilei java. security. policy la lansarea serverului, fie prin: System. setProperty()).

5. se înregistrează obiectul server la serverul de nume (rmiregister) sub numele dorit (java. rmi. Naming. bind() sau java. rmi. Naming. rebind()).
6. se implementează metodele apelabile de la distanţă (interfeţe la distanţă).

Scenariul utilizării mecanismului RMI pe un model de arhitectură client –server este prezentat în Figura 7.2.

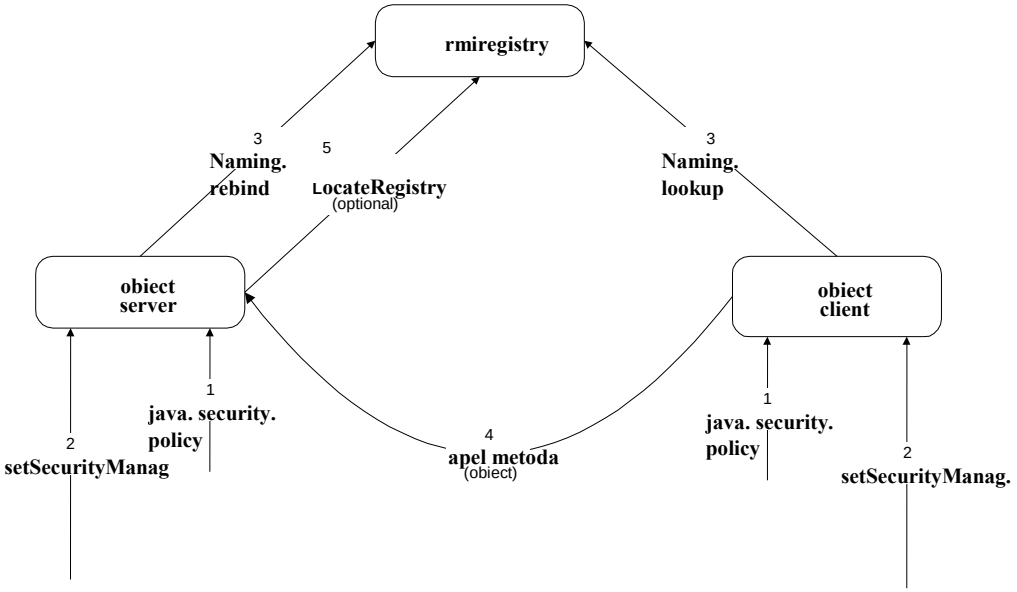


Figura 7.2.Scenariul utilizării RMI

7.2.4. Clase şi interfeţe RMI

Clase şi interfeţe client. În dezvoltarea părţii de client a unei aplicaţii se utilizează interfaţa *Remote* şi clasele *RemoteException şi Naming*. alături de o serie de excepţii derivate din RemoteException şi clasa RMISecurityManager utilizată la încărcarea claselor stub în aplicaţii.

Clasa Naming constituie mecanismul prin care clienţii pot obţine referiri la obiectele la distanta (ea este folosită şi de server pentru înregistrarea obiectelor cu rol de server la distanţă), localizarea obiectelor la distanţă are la bază mecanismul URL, clasa are doar metode statice şi nu defineşte constructori, metodele sale uzuale sunt lookup (), String(), bind(), rebind() unbind() utilizate pentru înregistrarea serverelor.

Clase şi interfeţe server. Exceptând clasa RMISecurityManager care se găseşte în pachetul *java. rmi*, toate celelalte clase şi interfeţe utilizate în programele server se găsesc în pachetul *java.rmi. server.*

Clasa remoteObject reimplementează metodele hashcode şi equals ale clasei Object pentru a permite memorarea referirilor de obiecte la distanţă în tabele de hasing şi compararea acestor referiri. Constructorii protected RemoteObject() şi protected RemoteObject(RemoteRef newref) creează un obiect la distanţă iniţializat cu referirea specificată.

Clasa RemoteServer este o superclasă pentru implementări de servere şi oferă o gamă variată de semantici pentru referirile la distanţă. Toate metodele clasei sunt statice. Metoda getClientHost apelata dintr-un fir de execuţie aflat în cursul tratării unui apel de metodă la distanţă returnează numele

calculatorului pe care rulează clientul, iar celelalte metode stabilesc identitatea fluxului utilizat la crearea unui jurnal de apeluri la server.

```
package java. rmi. server;
public abstract class RemoteServer extends RemoteObject{ protected
RemoteServer();
protected RemoteServer (remoteRef ref);
public static String getClientHost()
public static void setLog(java. io. OutputStream out); public static java. io.
PrintStream getLog(); }
```

Clasa UnicastRemoteObject. Această clasa derivă din RemoteServer şi oferă suport pentru referiri punct la punct la obiecte active bazat pe fluxuri TCP. Astfel serverele derivate vor fi nereplicate, iar referirile la ele vor fi doar pe durata de viaţă a procesului ce l-a creat.

```
package java. rmi. server;
public class UnicastRemoteObject extends RemoteServer {
protected UnicastRemoteObject()
protected UnicastRemoteObject(int port)
protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,
RMIServerSocketFactory ssf)
public Object clone()
public static RemoteStub exportObject(java. rmi. Remote obj)
public static Remote exportObject(java. rmi. Remote obj, int port)
public static Remote exportObject(Remote obj, int port, RMIClientSocketFactory
csf, RMIServerSocketFactory ssf)
public static boolean unexportObject(java. rmi. Remote obj, boolean force) }
```

Clasa **RMI SecurityManager**. În absenţa managerului de securitate, RMI nu va putea încărca clasele stub decât din sistemul local de fişiere (CLASSPATH). Managerul de securitate permite aplicaţiei să determine înaintea oricărei operaţii potenţial periculoase dacă ea se va efectua prin încărcătorul de clase sau din sistemul local, astfel aplicaţia poate autoriza sau dezautoriza operaţia. Dacă operaţia nu e permisă se va genera excepţia SecurityException.

Majoritatea metodelor au nume care încep cu check iar utilizarea lor se face după următoarea schemă:

```
SecurityManager security=System. getsecurityManager();
 if (security !=null){
Security. checkxxx(argument, …); }
```

Câteva metode din Clasa **RMISecurityManager** sunt:

- *public synchronized void checkAccess (Thread t) throws RMISecurityException* - stuburile nu pot modifica argumentul Thread
- *checkCreateClassLoader ()* – stuburile nu pot crea incarcătoare şi nici să execute metodele clasei ClassLoader
- *checkpropertiesAccess() (String key)* – stuburile nu pot accesa lista de proprietăţi a sistemului ci doar cele etichetate ca accesibile
- *checkListen(int port)* – stuburile nu pot asculta la nici un port
- *checkConnect(String host, int port, Object context)* clasele încărcate pot face conectări dacă sunt apelate prin transportul RMI.

Clasa **RMIClassLoader** – este un utilitar poate fi folosit de aplicaţii pentru a încarca clase de la un URL. Metodele sale sunt:

- *public static class loadClass(String name)* – foloseşte proprietatea java. rmi. server. codebase pentru a setermina URL –ul de la care se încarcă
- *public static synchronized Class LoadClass(URL codebase, String name)*

- *public static Object getSecurityContext(ClassLoader loader)* – returnează contextul de securitate

Procesele server trebuie să declare mediului de execuţie RMI localizarea claselor stuburi şi parametri/valori returnate ce vor fi disponibile clienţilor prin această proprietate.

Interfaţa LoaderHandler – este implementată de o clasa LoaderHandler, iar metodele sale sunt folosite de RMIClassLoader pentru efectuarea operaţiilor sale.

Clasa abstractă **RMI SocketFactory** – specifică modul în care transportul RMI trebuie să obţina socketuri. Specificarea clasei este următoarea:

```
package java. rmi. server;
public abstract class RMISocketFactory
implements RMIClientSocketFactory, RMIServerSocketFactory
{public abstract Socket createSocket(String host, int port)
public abstract ServerSocket createServerSocket(int port)
public static void setSocketFactory(RMISocketFactory fac)
public static RMISocketFactory getSocketFactory() {. . . }
public static void setFailureHandler(RMIFailureHandler fh) {. . . }
public static RMIFailureHandler getFailureHandler() {. . . } }
```

Serviciul de înregistrare a obiectelor (*Serviciul de nume*)

Pachetul **java. rmi. registry** conţine două interfeţe Registry şi RegistryHandler precum şi clasa LocateRegistry ce realizează serviciul de înregistrare şi regăsire a obiectelor la distanţă folosind nume simple. E permis ca fiecare process server să îşi definească propriul registru de obiecte sau să folosească un singur registru pentru un host. Registrul este un obiect la distanţă la rândul său. Interfaţa registry specifică metodele de *căutare, legare, relegare, ştergere şi listare* a conţinutului unui registru. Operaţii bind/rebind/unbind sunt premise doar dacă solicitanţii sunt pe acelaşi host ca şi serverul însă operaţia lookup este permisă de oriunde.

Clasa LocateRegistry - conţine metodele statice ce returnează o referinţă la un registru, o metodă de creare registru, referinţele returnate sunt referinţe la un stub la distanţă al registrului.

```
package java. rmi. registry;
public final class LocateRegistry { public static Registry getRegistry()
public static Registry getRegistry(int port)
public static Registry getRegistry(String host)
public static Registry getRegistry(String host, int port)
public static Registry getRegistry(String host, int port, RMIClientSocketFactory
csf)
public static Registry createRegistry(int port)
public static Registry createRegistry(int port, RMIClientSocketFactory csf,
RMIServerSocketFactory ssf) }
```

Metodele tip get returnează referiri al registrul hostului curent şi pentru portul specificat (portul pentru Registry este 1099). Metoda *CreateRegistry* creează şi exportă un registru pe hostul local, registru ce implementează un serviciu de nume simplu, în care numele unui obiect la distanţă (String) este asociat unei referinţe la distanţă. Legăturile astfel create sunt valabile doar pentru activarea curentă a registrului. Interfaţa *RegistryHandler* este utilizată pentru legătura cu o implementare particulară a serverului de nume astfel:

```
public interface RegistryHandler{
Registry regitsryStub(String host, int port)
Registry registryImpl(int port)  }
```

Clase şi interfeţe pentru *stuburi şi skeletoane*

Clasa RemoteStub – este superclasa comună stuburilor, ea oferă un cadru ce permite implementarea unor semantici diferite pentru referire la distanţă.

```
package java. rmi. server;
public abstract class RemoteStub extends java. rmi. RemoteObject { protected
RemoteStub() {. . . }
protected RemoteStub(RemoteRef ref) {. . . }
protected static void setRef(RemoteStub stub,  RemoteRef ref) {. . . } }
```

Clasa **RemoteCall** – este abstracţiunea utilizată de stuburi şi skeletoane pentru a transfera un apel la un obiect remote şi are următoarea specificare:

```
package java. rmi. server; import java. io. *;
public interface RemoteCall {
ObjectOutput getOutputStream() throws IOException;
void releaseOutputStream() throws IOException;
ObjectInput getInputStream() throws IOException;
void releaseInputStream() throws IOException;
ObjectOutput    getResultStream(boolean    success)    throws    IOException,
streamCorruptedException;
void executeCall() throws Exception;
void done() throws IOException; }
```

Primele metode referă fluxurile de obiecte prin care se realizează serializarea argumentelor, metoda GetResultStream returnează un flux de ieşire, după ce marchează în antet informaţie referitoare la succesul apelului, obţinerea fluxului cu rezultate trebuie să reuşească o singură dată pentru fiecare apel. Metoda ExecuteCall conţine ceea ce este necesar pentru efectuarea apelului, iar done eliberează resursele după terminarea apelului remote. Versiunile post JDK 1. 4. au depreciate această interfaţă în favoarea utilizării metodei invoke.

Interfaţa **RemoteRef** – defineşte un descriptor (handle) pentru un obiect la distanţă, fiecare stub integrează câte un exemplar RemoteRef ce conţine reprezentarea concretă a referinţei. Metoda NewCall permite crearea unui obiect de apel adecvat pentru un nou apel de metodă la distanţă obj. Tabloul de operaţii op conţine operaţiile disponibile asupra obiectului la distanţă iar num este un indice în acest tablou prin care este indicată operaţia apelului curent.

```
package java. rmi. server;
public interface RemoteRef extends java. io. Externalizable {
Object invoke(Remote obj, java. lang. reflect. Method method, Object[] params,
long opnum) throws Exception;
RemoteCall  newCall(RemoteObject  obj,  Operation[]  op,  int  opnum,   long  hash)
throws RemoteException;
void invoke(RemoteCall call) throws Exception;
void done(RemoteCall call) throws RemoteException;
String getRefClass(java. io. ObjectOutput out);
int remoteHashCode();
boolean remoteEquals(RemoteRef obj);
String remoteToString(); }
```

Interfaţa **ServerRef** reprezinta descriptorul de server pentru implementarea unui obiect la distanţă. Metoda exportObject caută sau creează un obiect stub client pentru implemenetarea obiectului Remote obj. Astfel server este obiectul server la distanţă pentru implementare poate fi identic cu obj), iar data conţine informaţiile necesare exportării obiectului.

```
package java. rmi. server;
public interface ServerRef extends RemoteRef {
RemoteStub exportObject(java. rmi. Remote obj, Object data) throws java. rmi.
RemoteException;
String getClientHost() throws ServerNotActiveException; }
```

Interfaţa **Skeleton** este utilizată în implementarea skeletoanelor generate de compilatorul rmic. Metoda dispatch deordonanţează argumentele din fluxul de intrare obţinut de la obiectul call, apelează metoda opnum pe implementarea efectiva a obiectului obj şi generează valoarea returnată.

```
package java. rmi. server;
public interface Skeleton { void dispatch(Remote obj, RemoteCall call, int opnum,
long hash) throws Exception;
Operation[] getOperations(); }

Clasa Operation – menţine descrierea unei metode pentru un obiect la distanţă
public class Operation{ public Operation (String op);
public String getOperation(); public String toString(); }
```

Securitatea RMI

RMI pune la dispoziţie pentru securizarea canalelor de comunicaţie între client şi server un mecanism de izolare a implementărilor aduse intr-un 'sand box' securizat pentru a proteja sistemul de posibile atacuri efectuate de clienţi nesiguri. Este important a se defini nevoile de securitate pentru a stabili reguli stricte ce se vor aplica pentru toţi clienţii. S-ar putea să fie nevoie a se securiza un canal de comunicaţie între client şi server.

RMI permite furnizarea unui constructor de socket-uri care permite crearea unor socket-uri de diferite tipuri, inclusiv *socket-uri criptate*, care vor fi folosite la comunicaţie între client şi server. Începând cu JDK 1. 2 se vor putea specifica cerinţele pentru serviciile oferite pentru socket-urile server-ului. Aceasta nouă tehnică este utilă în applet-uri, unde majoritatea browser-elor refuză permisiunea de setare a unui constructor de tipuri de socket-uri.

Clasele aduse remote prezintă alte probleme de securitate, securizarea acestora este realizată via un obiect de tip *SecurityManager*, care va superviza orice acţiune ce implică un nivel de securitate mai deosebit, cum ar fi deschiderea de fişiere, stabilirea conexiunilor pentru reţea etc. RMI furnizează un tip RMISecurityManager, care e la fel de restrictiv ca acele tipuri folosite pentru securizarea applet-urilor, astfel se previne ca implementările aduse remote să poată citi date sau scrie date locale, sau să creeze conexiuni spre alte sisteme din spatele firewall-urilor. Se poate de asemenea scrie şi instala un obiect propriu de securitate pentru a forţa diferite politici de securitate specifice.

Nu este însă necesar şi nici recomandabil să se altereze politica implicită a JDK, ci este preferabil să se definească o politică particulară aplicaţiei ce ridică probleme. Pentru a lansa în execuţie o aplicaţie Java cu altă politică decât cea implicită se poate folosi un fişier de configurare care să definească noua politică. Apoi, maşina virtuală trebuie instruită să folosească noua politică plasând numele fişierului respectiv în proprietatea globală *java. security. policy.*

Un asemenea fişier ar putea avea următorul conţinut:

```
    grant {
    permission java. io. filePermission "/tmp/*", "read", "write";
    permission java. net. SocketPermission "somehost. somedomain. com:999",
    "connect";
    permission java. net. SocketPermission "*:1024-65535", "connect, request";
permission java. net. SocketPermission "*:80", "connect";  };
```

8.2.5. Mecanisme complementare

a)Adnotarea şi rezolvarea claselor în RMI. Adnotarea claselor denotă mecanismul prin care în fluxul de date serializat *se adaugă şi o adresă de unde se poate obţine la nevoie codul binar Java al clasei respective,* mecanism fundamental pentru implementarea transferului de comportament în RMI.

Pentru a realiza adnotarea unei clase la scriere (serializare) în loc să se utilizeze un obiect instanţiat direct din ObjectOutputStream se utilizează un obiect instanţiat din clasa derivată MarshalOutputStream, în care se defineşte metoda *annotateClass().* Pentru a încărca clasele adnotate

(operaţia inversă serializării) se utilizează un obiect instanţiat din MarshalInputStream care derivă din ObjectInputStream şi care defineşte metoda resolveClass(). Adresa cu care se face adnotarea se transmite sub forma unui URL.

La împachetarea argumentelor pentru RMI (marshalling) se utilizează pentru fiecare obiect argument metoda annotateClass() care testează dacă *o clasă este locală sau a fost adusă de un încărcător (loader) de la distanţă.* În acest caz (de un URLClassLoader spre exemplu), se interoghează încărcătorul respectiv asupra adresei de origine şi se adnotează cu aceasta clasa, în caz contrar, RMI nu poate determina adresa pe baza originii clasei . Pentru a adnota astfel de clase RMI are nevoie de o adresă dată explicit prin intermediul proprietăţii *java. rmi. server. codebase*, proprietate ce permite *chiar specificarea mai multor adrese alternative*. Dacă această proprietate este însă vidă, atunci clasele nu vor fi adnotate, iar cel care le receptează trebuie să determine prin alte mijloace adresa de la care le poate încărca.

Rezolvarea claselor în RMI. « Rezolvarea » unei clase înseamnă obţinerea codului corespunzător metodelor sale. La despachetarea argumentelor (unmarshalling) se apelează pentru fiecare descriptor de clasă metoda resolveClass() din MarshalInputStream, responsabilă cu încărcarea claselor necesare pentru instanţierea obiectului.

Înainte de a preciza strategia de rezolvare a claselor, este necesară specificarea modului în care se poate obţine codul binar al clasei, astfel există trei metode posibile:

- pe baza adnotării aferente clasei transmisă prin RMI Wire Protocol
- pe baza unei proprietăţi globale pentru toate clasele recepţionate fără adnotare
- prin intermediul unui încărcător special.

Proprietatea globală care poate fi interogată este tot java. rmi. server. codebase. Se observă că această proprietate oferă o adnotare automată a claselor ce nu prezintă una explicită atât la transmiterea cât şi la recepţionarea fluxului serializat, mecanism numit *adnotare implicită.* Încărcătorul special menţionat poartă denumirea de încărcător contextual (context class loader). Acest mecanism este specific Java 2 şi este foarte flexibil deoarece pentru fiecare fir de execuţie se poate specifica un încărcător contextual diferit. La instanţierea unui nou fir de execuţie, acesta moşteneşte încărcătorul contextual al părintelui.

Etapele de rezolvare a claselor sunt:

- modalitatea “clasică” de rezolvare care se aplică tuturor claselor serializate, presupune căutarea în stivă (prin intermediul unei metode native) a unei clase cu încărcătorul definitor (defining class loader) nenul. Dacă există, acest încărcător este invocat pentru a rezolva clasa serializată.
- se testează dacă clasa este adnotată, dacă nu este se va folosi drept adnotare implicită java. rmi. server. codebase (comportament ce poate fi forţat chiar dacă clasa este adnotată în fluxul serializat, cu condiţia ca proprietatea globală java. rmi. server. useCodebaseOnly să aibă valoarea true), insa pentru ca adnotarea să fie luată în consideraţie este necesar să fie activ un manager de securitate.
- se presupune activ un manager de securitate

Ordinea în care se încearcă rezolvarea este întotdeauna: adnotarea şi apoi încărcătorul contextual. Evident, în absenţa unui manager de securitate se va încerca direct încărcătorul contextual, iar dacă acesta eşuează se va afişa un mesaj explicit ce avertizează utilizatorul că adnotarea nu a fost luată în considerare.

Astfel proprietatea *java. rmi. server. codebase* poate fi utilizată pentru adnotare atât de către cel care transmite clasele cât şi de cel care le recepţionează, insa este necesara stabilirea unei politici adecvate în acest sens.

Să presupunem că dorim să realizăm o aplicaţie, în care un server transmite un comportament care va fi executat de către clienţi. Există trei posibilităţi şi anume:

- setarea proprietăţii codebase doar la client, astfel clasele vor sosi fără vreo adnotare la acesta şi se va folosi adnotarea implicită.
- setarea proprietăţii codebase doar la server, atunci clasele vor sosi direct adnotate.

- setarea proprietăţii codebase şi la client şi la server, atunci clasele vor sosi adnotate de pe server, iar adnotarea implicită a clientului va fi ignorată. Se observă că în acest scenariu transferul va funcţiona indiferent unde se specifică proprietatea codebase.

Să presupunem însă că dorim să ne conectăm cu clientul la mai multe servere simultan şi că nu toate serverele au codul comportamentului la aceeaşi adresă. În această ipoteză clientul trebuie să primească cod de la fiecare server, iar adnotările trebuie să fie specifice fiecărui server, caz în care este utila a doua soluţie, în care fiecare server va adnota corespunzător fluxul său de date, astfel încât clientul să poată obţine fiecare comportament de la sursa corespunzătoare.

b)Flux de date, transfer de comportament

O aplicaţie care doreşte să transfere comportament va utiliza două fluxuri de informaţie: fluxul de date (transmis prin RMI Wire Protocol) şi fluxul de cod binar Java.

Fluxul de date: RMI Wire Protocol. Funcţionarea RMI se bazează pe protocolul RMI Transport Protocol care realizează implementarea fluxului de date între apelant şi apelat (protocol de nivel aplicaţie), el încapsulând celelalte protocoale folosite. Protocolul în sine este suficient de flexibil ca să pară transparent pentru programatori, el implementează trei tipuri de conexiuni: directe, multiplexate şi încapsulate, în mod implicit atât conexiunile directe cât şi cele multiplexate se efectuează peste TCP.

Conexiunile directe pot fi utilizate atunci când nu există restricţii de securitate la nivelul maşinii virtuale a serverului şi nici la nivelul reţelei. Conexiunile multiplexate sunt utile atunci când se doreşte exportarea unor obiecte dintr-un applet (exportarea este acţiunea prin care un obiect devine invocabil de la distanţă).

Conexiunile încapsulate se utilizează atunci când clientul se află într-un intranet protejat de un firewall care limitează accesul către exterior. Pentru conexiunile încapsulate clientul împachetează apelul într-o cerere HTTP POST, răspunsul obiectului apelat fiind împachetat la rândul său într-un răspuns HTTP.

Pentru a transmite cererea sunt disponibile două metode:

- Acces direct - se încearcă adresarea acesteia direct către portul pe care ascultă obiectul de pe server, dacă firewall-ul lasă să treacă cereri HTTP către porturi arbitrare, atunci această metodă va reuşi, iar obiectul va fi apelat direct. În caz contrar, se trimite cererea către portul 80 al serverului, în speranţa că firewall-ul permite conexiuni către porturi HTTP standard.
- Acces indirect – e necesar ca pe maşina pe care se găseşte obiectul apelat să ruleze un server Web care să poată executa un anume program CGI (java-rmi. cgi). Acest program va citi antetul cererii POST şi pe baza acestuia va înainta cererea către portul pe care ascultă obiectul apelat.

Prin conexiunea stabilită prin una din aceste metode sunt transmise invocările la distanţă şi obţinute rezultatele. Prin intermediul primelor două tipuri de conexiune se pot transmite mai multe apeluri, însă conexiunea încapsulată permite un singur apel, după care trebuie efectuată o nouă conexiune.

Flux de cod (comportament). Pentru a încapsula fluxul de cod binar se pot folosi protocoalele Internet pentru transferul de fişiere suportate de maşina virtuală Java (FTP, HTTP). Nu este nevoie de un server Web pentru a testa transferul de cod, fiind posibilă folosirea oricărui protocol pentru care există un handler instalat, bazat pe faptul că RMI foloseşte un URLClassLoader pentru a transfera codul binar al claselor ce prezintă adnotare.

c) Redefinirea socketurilor RMI

O facilitate deosebit de utilă este aceea de a permite utilizatorului să redefinească mecanismul de generare al socketurilor folosite de implementarea RMI, fapt ce permite integrarea de capabilităţi speciale pentru situaţiile în care *se impune o anumită politică de securitate în reţea* şi trebuie să existe control asupra porturilor utilizate de RMI. Aceasta este realizata prin modificarea fluxului de date

transmis (cum ar fi utilizarea RMI peste protocolul de transmisie securizata SSL -Secure Sockets Layer) asigurând astfel prin criptare securitatea comunicaţiilor prin utilizarea unui mecanism specific numit Custom RMI Socket Factory.

RMISocketFactory face posibilă construirea unei "fabrici" care să producă socketuri, însă odată instanţiata "fabrica" aceasta putea produce numai socketuri de acelaşi tip. Prin introducerea clasei java. *rmi. server. SocketType* se oferă posibilitatea construirii unui RMISocketFactory care poate genera cel *mai potrivit tip de socket pentru un anumit obiect.* Practic, se oferă utilizatorului RMI posibilitatea de a redefini fabrica ce produce soclurile utilizate de RMI, redefinire introdusă încă din JDK 1. 1, însă o serie de restricţii destul de severe au făcut ca această facilitate să nu poată fi exploatată corespunzător în JDK 1. 1, astfel:

- aceeaşi fabrică de socluri era necesar a fi folosită pentru toate obiectele invocabile de la distanţă,
- era necesar să existe un registru RMI (rmiregistry) pentru fiecare tip de socluri, iar
- clasele fabricii de socluri nu puteau fi aduse de la distanţă ci era obligatoriu să fie încărcate local.

Limitările prezentate au dispărut ,astfel mecanismul în sine este analog cu fabricile de socluri « clasice » din pachetul java.net, concret, trebuie realizată o implementare următoarelor două interfeţe:

```
   public interface RMIClientSocketFactory { public Socket createSocket(String host,
int port) throws IOException; }
   public      interface      RMIServerSocketFactory      {public      ServerSocket
createServerSocket(int port) throws IOException; }
```

Tipul soclurilor (tip client pentru cel care iniţiază o conexiune, tip server ascultă în aşteptarea unei conexiuni) utilizate de un obiect invocabil la distanţă se stabileşte la momentul exportării obiectului. Presupunând că obiectul este derivat din UnicastRemoteObject, va fi suficient să-l construim prin intermediul unui constructor care primeşte ca argumente fabricile de socluri: *protected UnicastRemoteObject(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)* .

Ambele tipuri de socluri se transmit ca parametri. Fabrica de socluri server va fi folosită de către obiect pentru a instanţia socluri pe care se vor primi conexiuni de la clienţi, iar fabrica de socluri client nu e folosită ca atare de obiectul server, ci se va transmite clienţilor pentru ca aceştia să se poată conecta la server. Putem imagina un client care contactează un server prin RMI « obişnuit », stabileşte de comun acord cu acesta ca sesiunea să se desfăşoare criptat, după care utilizează o fabrică de socluri cu criptare pentru restul comunicaţiei, iar fabrica ce generează soclurile cu criptare poate fi transmisă de la server la client pe parcursul execuţiei acestuia.

d)Apeluri tip Callback

În programarea concurentă, noţiunea de invocare la distanţa integrează un aspect suplimentar: posibilitatea ca maşina server să « întoarcă un răspuns », adică să determine executarea unei metode pe maşina client, facilitate suplimentară ce poartă numele de apel invers (callback). Pentru a putea realiza apelurile inverse este necesar să se definească o nouă interfaţă la distanţă şi să se implementeze aceasta interfaţă de către client, iar interfaţa la distanţă implementată de server va trebui să prevadă o metodă prin care clienţii îşi pot înregistra la server interesul pentru a fi anunţaţi de apariţia anumitor modificări în starea serverului. Principial, lucrurile decurg analog ca la invocarea la distanţă a unei metode. Fie metC metoda care trebuie invocată pe maşina client din cadrul metodei metS. Este evident necesar ca pe maşina client să fie creat un obiect ObC de tipul clasei care include metoda metC, iar maşina server sa primească o referinţă la acest obiect, pentru ca (pe baza acestei referinţe) să poată invoca metoda metC, ce va fi executată pe maşina client. Implementarea apelurilor inverse va fi mai simpla, deoarece legătura între maşini este deja stabilită.

Pentru realizarea apelurilor inverse, trebuie întreprinse doar următoarele două acţiuni: 1) crearea obiectului şi pregătirea lui pentru a fi exportat; 2) transmiterea obiectului maşinii server. Prima acţiune presupune ca ObC este o instanţă a unei clase ce extinde interfaţa Remote. Pregătirea sa pentru

export este realizata prin invocarea următoarei metode publice şi statice a clasei UnicastRemoteObject: RemoteStub exportObject(Remote ObC) throws RemoteException. A doua acţiune poate fi realizată simplu, comunicând serverului obiectul ObC prin includerea lui ca argument la invocarea unei metode prin intermediul referinţei la obiectul ObS. Se presupune că obiectul ObC este de tipul unei clase serializabile.

e) Obiecte activabile

Mecanismul RMI se îmbogăţeşte cu obiecte activabile la distanţă. Acestea reflecta un mecanism prin intermediul căruia se pot activa obiecte server *doar atunci când este nevoie de ele şi* se folosesc identificatori persistenţi pentru a putea referi obiectele şi atunci când acestea sunt inactive, aceasta conduce evident la o folosire mai eficientă a memoriei pe maşina server.

De lansarea în execuţie a acestor obiecte activabile este responsabil un demon (rmid) ce lansează maşini virtuale Java în care rulează efectiv obiectele. Pentru a nu crea un număr excesiv de maşini virtuale şi pentru a permite cooperarea între obiectele server, acestea pot fi adunate în grupuri de activare; toate obiectele din acelaşi grup fiind executate în aceeaşi maşină virtuală. Prin introducerea clasei *java. RMI. activation. Activatable* şi a demonului RMI, rmid, obiectele remote pot fi create şi executate atunci când este nevoie de ele, programele trebuie doar să înregistreze informaţiile de implementare despre obiectele remote şi demonul rmid va furniza instanţa obiectului atunci când va fi nevoie de ea.

7.3.Exemple

7.3.1. Implementarea unui sistem simplu de tip client-server folosind tehnologia RMI

Implementare client

```
import java.rmi.*;
import java.rmi.server.*;

public class SampleClient
{
   public static void main(String[]  args)
   {
      // set the security manager for the client
      System.setSecurityManager(new RMISecurityManager());
      //get the remote object from the registry
      try
        {
          System.out.println("Security Manager loaded");
          String url = "//localhost/SAMPLE-SERVER";
          SampleServer remoteObject = (SampleServer)Naming.lookup(url);
          System.out.println("Got remote object");
          //narrow the object down to a specific one
       //System.out.println("Location: " + System.getProperty("LOCATION"));
          // make the invocation

          System.out.println(" 1 + 2 = " +
               remoteObject.sum(1,2) );
        }
      catch (RemoteException exc)
        {
          System.out.println("Error in lookup: " + exc.toString());
        }
      catch (java.net.MalformedURLException exc)
        {
          System.out.println("Malformed URL: " + exc.toString());
        }
      catch (java.rmi.NotBoundException exc)
        {
          System.out.println("NotBound: " + exc.toString());}}}
```

Implementare Server –interfața și implementarea sa

```
import java.rmi.*;
public interface SampleServer extends Remote
{
  public int sum(int a,int b)throws RemoteException;}
```

Implementarea interfeței serverului

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class SampleServerImpl extends UnicastRemoteObject
                              implements SampleServer
{
  SampleServerImpl() throws RemoteException
  {
     super(); }

  public int sum(int a,int b) throws RemoteException
  {
     return a + b;}

  public static void main(String args[])
  {
    //set the security manager
    try
      {
        System.setSecurityManager(new RMISecurityManager());

        //create a local instance of the object
        SampleServerImpl Server = new SampleServerImpl();

        //put the local instance in the registry
        Naming.rebind("SAMPLE-SERVER" , Server);

        System.out.println("Server waiting....."); }
    catch (java.net.MalformedURLException me)
      {
         System.out.println("Malformed URL: " + me.toString()); }

    catch (RemoteException re)
      {
         System.out.println("Remote exception: " + re.toString()); }}}
```

7.3.2.Se vor testa exemplele ce ilustrează modul de utilizare a tehnologiei RMI, exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/07_RMI

7.4.Întrebări teoretice

7.4.1.Descrieți arhitectura RMI specificând funcționalitățile asigurate de stub/skeleton.

7.4.2. Ce reprezintă serializarea obiectelor și ce rol joacă în tehnologia RMI

7.4.3.Ce reprezintă Rmiregistry și care este utilitatea sa?Descrieți fluxul funcțional pe o aplicație simplă de tip calculator cu implementare folosind tehnologia RMI.

7.4.4.Explicați modelul de securitate RMI .

7.4.5.Identificați și explicați succint utilitatea pentru două din mecanismele complementare RMI ce extind contextul de bază al tehnologiei oferind funcționalități evoluate pentru dezvoltarea unor aplicații complexe.

7.5.Probleme propuse

7.5.1.Să se implementeze un sistem client-server care să ofere funcționalitatea unei camere de chat mai multor utilizatori. Serverul gestionează o cameră de chat acceptând mai multe conexiuni simultane de la utilizatorii care vor să discute. Utilizatorii trebuie să se autentifice printr-un nume unic și o parolă. Serverul va primi mesaje de la utilizatorii conectați și va trimite imediat tuturor clienților conectați mesajele primite. Clientul se conectează la server și oferă utilizatorului o interfata prin care poate să trimită mesaje spre camera de discuții. Un mesaj trimis de un utilizator va apărea simultan in interfața oferită tuturor utilizatorilor, inclusiv a celui care a scris initial mesajul.

7.5.2.Să se realizeze unui program de transfer de fișiere prin rețea folosind tehnologia RMI. Astfel, un server concurent va pune la dispozitie un arbore de directoare și fișiere din care poate trimite, la cerere, fișiere către clienti .Clientul se conectează la server și solicită fisiere. Pentru aceasta vor folosi comenzi speciale de tipul get nume_fisier - cere fișierul din directorul curent. Clientul îl va primi și îl va salva în directorul sau curent.Alte comenzi ce se cer implementate sunt :put nume fisier - trimite spre server fisierul,dir - cere lista de fisiere de la server,cd - schimba directorul curent pe server. Va trebui să accepte forma cd ..care se referă la directorul părinte. Orice erori de protocol (fișierul nu există, comandă incorectă) vor fi tratate în server și înțelese de client.

7.5.3.Se dorește implementarea unui sistem de discuții între doi utilizatori format din două componente simetrice care se comportă alternativ ca server respectiv client. (modelul peer to peer).Una din componente (A) initiază comunicarea, conectându-se la adresa celeilalte (B) și specificând un nume de utilizator cu care doreste să vorbească. Componenta B poate refuza comunicarea pe următoarele motive: utilizatorul nu e de față ,utilizatorul nu există,utilizatorul a blocat accesul pentru utilizatorul apelant (acesta din urmă este pe lista celor nedoriți de către destinatar)sau utilizatorul a răspuns cu comanda "Ocupat, reveniti mai tarziu".După stabilirea conexiunii, cei doi utilizatori își vor putea trimite unul altuia mesaje. Modul de comunicare între procesele de pe acelasi calculator poate fi ales.

7.5.4.Realizarea unui program de sincronizare a unui arbore de fișiere local cu un arbore similar de la distanță. Se consideră următoarea situație: pe un calculator server există un arbore de directoare și fisiere care conțin tot timpul ultima versiune a unui sistem software. Mai multe alte calculatoare doresc sa aibă imagini oglindă ale distribuției (mirrors) pe care să le mențină cât mai la zi. Pentru aceasta, pe server există un program server care gestioneaza distribuția, iar pe fiecare calculator "mirror" cate un program client. Clientul se conectează la server și realizează aducerea la zi a fișierelor de pe server. Criteriile de aducere la zi vor fi: data ultimei modificări a fisierelor,dimensiunea fișierelor,ștergerea fișierelor locale care nu mai există pe server,aducerea fișierelor noi de la server.

7.5.5.Se va realiza un program care permite lansarea de comenzi la distanță. Serverul primește de la client, în mod succesiv, comenzi pe care le execută local. Rezultatul execuției (iesirea standard) este trimis către client ca și raspuns. La primirea comenzii exit termină conexiunea. Clientul se conectează la server și trimite comenzi către acesta. Primește rezultatul îl afișează pe ecran, respectiv acceptă și înțelege mesajele de eroare trimise de server. Se vor trata explicit erori de tipul:comandă inexistentă,comanda executată a returnat cod de eroare,timp prea lung de execție a comenzii (timeout -dacă a trecut un timp prea lung de la lansarea comenzii, iar serverul nu a trimis încă rezultat).

7.6. Referințe bibliografice

1. Plugin RMI –Eclipse pentru compilare , execuție și analiză :https://marketplace.eclipse.org/content/rmi-plug-eclipse
2. Ghid RMI https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/
3. RMI în Java 7 :https://www.sciencedirect.com/topics/computer-science/remote-method-invocation
4. Tutorial RMI și exemple : http://aipi2015.andreirosucojocaru.ro/laboratoare/laborator03

8.Sincronizare și coordonare în sisteme distribuite folosind algoritmi distribuiți

Una din problemele fundamentale ce apare în momentul dezvoltării unui sistem alcătuit din procese multiple independente o constituie certitudinea că procesele realizează execuția corect, la momente de timp potrivite, fapt ce presupune ca acestea să își sincronizeze şi coordoneze acţiunile, motiv pentru care la nivelul unui sistem distribuit apare o coordonată suplimentară și anume timpul. Sincronizarea şi coordonarea joacă un rol foarte important în majoritatea sistemelor distribuite.Prin *coordonare* se face referire la corelarea acțiunilor proceselor separate şi asigurarea unor mecanisme de comunicare la nivel global. *Sincronizarea* este similară coordonării însă integrează și factorul timp, referind ordinea evenimentelor şi execuţia instrucţiunilor raportată la timp. Un exemplu al sincronizării este ordonarea evenimentelor distribuite dintr-un fişier de tip log file şi certitudinea că un proces realizează o anumită acţiune la un anumit moment de timp. Un exemplu de coordonare poate fi considerat agreementul (consensul) la care procesele ajung în momentul identificării și alegerii acțiunilor ce urmează a fi executate de către unul dintre ele sau în manieră coordonată.

8.1.Obiective

- Modele de timp logic în sisteme distribuite
- Studiul şi implementarea unor algoritmi de coordonare ,sincronizare a proceselor şi excludere mutuală distribuită în accesarea de resurse

8.2. Concepte

Problema timpului dintr-un sistem nedistribuit este una trivială – existând *un singur ceas comun al sistemului,* caz în care toate procesele văd acelaşi *timp global*. Pe de altă parte, într-un sistem distribuit, fiecare calculator are propriul său semnal de ceas. Datorită faptului că nici un ceas nu poate fi perfect, fiecare dintre aceste semnale au propriile lor întârzieri (aşa numitele *clock skew- variația ceasurilor)* ce referă faptul că semnalul unui ceas ajunge la diferitele componente, în diferite momente de timp. Aceste întârzieri au ca efect variații uneori nepermise pentru ceasurile diverselor sisteme de calcul și astfel acestea pot ajunge în stare desincronizată.

Există diverse concepte referitoare la timp, cu relevanţă într-un sistem distribuit. În primul rând, un ceas intern ţine evidenţe care pot fi translatate în timp fizic (ore, minute, secunde, etc.). Acest timp fizic poate fi global sau local. Timpul global este un timp universal, acelaşi pentru toate componentele, şi este bazat în general pe o formă de timp absolut (timpul global cel mai precis este UTC (Coordinated Universal Time). Pe lângă timpul global, procesele pot avea şi un timp local, caz în care timpul este relevant doar proceselor în mod individual. Timpul local poate să fie bazat pe *ceasuri fizice sau logice*, categorii ce vor fi detaliate în continuare.

În cadrul algoritmilor distribuiţi, în general, sunt utilizate două tipuri de modele de timp şi anume un model sincron, timpul necesar realizării tuturor sarcinilor, întârzierile din timpul comunicării şi diferenţele între ceasul fiecărui nod, sunt cu valori limitate, cunoscute, respectiv modelul asincron fără limite de timp. Sistemele distribuite cele mai apropiate de realitate sunt asincrone, însă este mult mai simplă dezvoltarea unui algoritm distribuit pentru un sistem sincron.

8.2.1.Ceasuri fizice și ceasuri logice în sisteme distribuite

Semnalele de ceas fizice ţin evidenţa timpului fizic. În sistemele distribuite care se bazează pe timpul efectiv, este necesară menţinerea semnalului de ceas al fiecărui calculator, în stare sincronizată. Ele pot fi sincronizate cu timpul global (sincronizare externă), sau unele cu altele (sincronizare internă).

Algoritmul Cristian (Cristian`s Algorithm) şi Network Time Protocol (NTP) sunt exemple de algoritmi dezvoltaţi pentru sincronizarea semnalelor cu o sursă de timp externă şi totodată globală (de obicei UTC). *Algoritmul Berkeley (The Berkeley Algorithm)* este un exemplu de algoritm care permite sincronizarea semnalelor de ceas în mod intern.

Pentru multe aplicaţii, *ordonarea relativă* a evenimentelor este mult mai importantă decât timpul fizic efectiv. Într-un singur proces, ordonarea evenimentelor (de exemplu, schimbările de stare) este trivială. Într-un sistem distribuit, pe lângă ordonarea locală a evenimentelor, toate procesele trebuie să ajungă la consens cu privire la ordonarea evenimentelor asociate cauzal (de exemplu, trimiterea şi primirea unui mesaj singular).
Fiind dat un sistem ce conţine *N procese* p_i*, i luând valori între {1,...,N},* se defineşte o ordonare locală a evenimentelor (notată →) ca o relaţie binară, astfel încât, dacă p_i observă evenimentul e înainte de e', astfel avem $e \rightarrow_i e'$. Bazat pe această ordonare locală, definim o *ordonare globală* ca o relaţie de tipul *happened before* →, definiţia dată de Lamport fiind următoarea:

Relaţia → este astfel încât:

1. $e \rightarrow_i e'$ implică $e \rightarrow e'$,
2. pentru fiecare mesaj m, *send*(*m*) → *receive*(*m*), şi
3. $e \rightarrow e'$ şi $e' \rightarrow e''$ implică $e \rightarrow e''$ (tranzitivitatea).

Relaţia → este asimilată unei relaţii de ordonare parţială (îi lipseşte reflexivitatea). Dacă a → b, atunci spunem că *a îl afectează cauzal pe b*. Considerăm că evenimentele neordonate sunt *concurente*.

Ca un exemplu, considerăm Figura 1. Avem următoarele relaţii cauzale:

$E11 \rightarrow E12, E13, E14, E23, E24, ...$
$E21 \rightarrow E22, E23, E24, E13, E14, ...$

Mai mult, următoarele evenimente sunt concurente $E_{11}||E_{21}$, $E_{12}||E_{22}$, $E_{13}||E_{23}$, $E_{11}||E_{22}$, $E_{13}||E_{24}$, $E_{14}||E_{23}$.

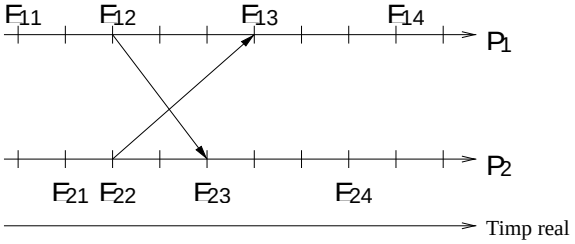


Figura 8.1. Ordonarea evenimentelor

8.2.2.Ceasuri logice Lamport

Ceasurile logice Lamport pot fi implementate sub forma unor numărătoare care calculează local relaţia de tipul *happened-before* →. Aceasta înseamnă că, fiecare proces p_i *reţine un ceas logic* L_i. Fiind dat un asemenea semnal, $L_i(e)$ denotă un timestamp de tipul Lamport al evenimentului e la p_i şi $L(e)$ denotă un timestamp al evenimentului e la procesul la care acesta a apărut.
Evoluţia la nivel de proces este următoarea :

1. Înainte de a atribui un timestamp unui eveniment local, un proces p_i execută $L_i := L_i + 1$.
2. De fiecare dată când un mesaj m este trimis de la p_i la p_j:

- Procesul p_i execută $L_i := L_i + 1$ şi trimite noul L_i *cu m*.
- Procesul p_j recepţionează L_i *cu m* şi execută $L_j := \max(L_j, L_i) + 1$. *receive*(*m*) este notat cu noul L_j.

În această schemă, $a \rightarrow b$ implică $L(a) < L(b)$, dar $L(a) < L(b)$ nu implică în mod necesar $a \rightarrow b$. Ca un exemplu, considerăm Figura 2. În această figură $E_{12} \rightarrow E_{23}$ şi $L_1(E_{12}) < L_2(E_{23})$, totodată avem şi $E_{13} \rightarrow E_{24}$ cât timp $L_1(E_{13}) < L_2(E_{24})$.

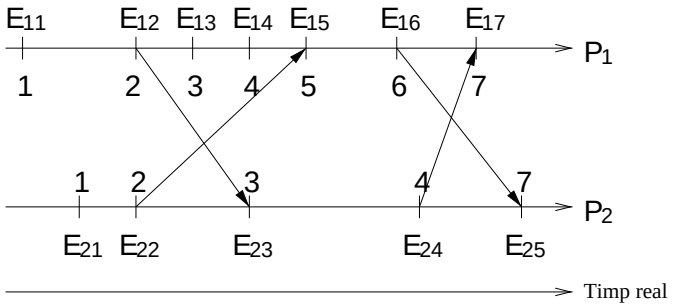


Figura 8.2. Exemplu de utilizare a ceasurilor Lamport

În unele situaţii, de exemplu, implementarea lacătelor distribuite pentru algoritmi de excludere mutual distribuită, o ordonare parţială a evenimentelor nu este suficientă şi deci, este necesară o ordonare totală. În aceste cazuri, ordonarea parţială poate fi completată pentru obţinerea unei ordonări totale, prin includerea identificatorilor de procese. Dându-se următoarele stampe de timp *locale* $L_i(e)$ şi $L_j(e')$, definim *stampe de timp globale, a*poi utilizăm ordonarea standard lexicografică.

8.2.3.Ceasuri logice vectoriale

Principala deficienţă a ceasurilor logice Lamport este faptul că $L(a) < L(b)$ nu implică $a \rightarrow b$; prin urmare, nu putem deduce dependenţe cauzale din timestamps. De exemplu, în Figura 3, avem $L_1(E_{11}) < L_3(E_{33})$, dar E_{11} 6→ E_{33}.
Esenţa problemei este reprezentată de înaintarea independentă sau prin mesaje a ceasurilor ,fără menţinerea unui istoric al acestei înaintări.Problema poate fi rezolvată prin trecerea de la ceasuri scalare la vectori de ceasuri, unde fiecare proces reţine un vector V_i.

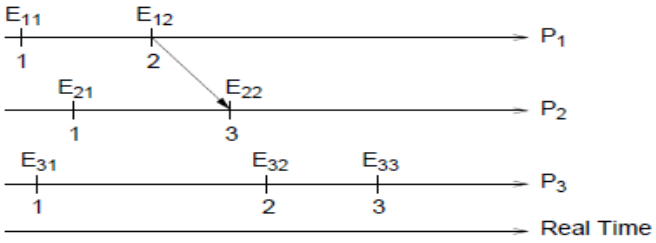


Figura 8.3. Exemplu al lipsei de cauzalitate cu ceasuri Lamport

V_i *este un vector de mărime N,* unde N este numărul de procese. Componenta $V_i[j]$ conţine informaţiile pe care le deţine procesul p_i *referitoare la semnalul* p_j. Iniţial, avem $V_i[j] := 0$ pentru $i,j \in \{1,...,N\}$. Semnalele înaintează astfel:

1. Înainte ca p_i să atribuie un timestamp unui eveniment, execută $V_i[i] := V_i[i] + 1$.
2. De fiecare dată când un mesaj m este trimis de la p_i la p_j:
   Procesul p_i execută $V_i[i] := V_i[i] + 1$ şi trimite V_i cu *m.*
   Procesul p_j primeşte V_i cu *m şi* adună semnalele de vector V_i şi V_j astfel:
   $Vj[k]:=\max(Vj[k],Vi[k]) + 1$ *,dacă* $j = k$
   $Vj[k]:= \max(Vj[k],Vi[k])$ *, dacă* $j \# k$

Astfel se asigură faptul că tot ceea ce se întâmplă ulterior la procesul p_j , este acum în relaţie cauzală cu tot ceea ce s-a întâmplat la p_j.În această abordare , avem pentru oricare i,j, $V_i[i] \geq V_j[i]$ (de

exemplu, p_i are întotdeauna cea mai nouă versiune a propriului ceas); mai mult, $a \rightarrow b$ *dacă şi numai dacă* $V(a) < V(b)$.De exemplu, considerăm notaţiile din diagrama reprezentată în Figura 4. Fiecare eveniment este notat atât cu valoarea vectorului de ceas propriu (tripletul), cât şi valoarea corespondentă a unui ceas scalar Lamport.

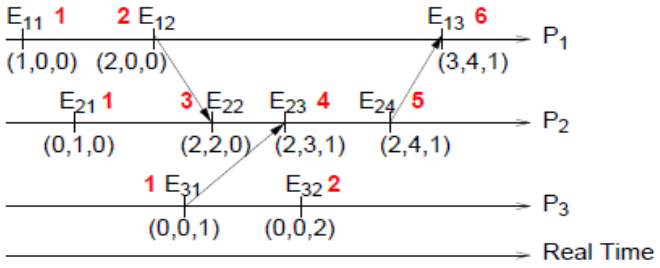


Figura 8.4. Exemplu ceasuri vectoriale

8.2.4.Algoritmi pentru ceasuri logice

Pentru cei doi algoritmi , este valid un set de *ipoteze generale*, şi anume :

1: procesele din sistemul distribuit comunică prin schimb de mesaje (în sens abstract, pot fi mesaje TCP, UDP RMI etc.)

2: în sistem fiecare process cunoaşte procesele cărora le trimite mesaje şi de la care poate primi mesaje.

3: transferal de mesaje poate fi sincron. Astfel este cunoscută o limită superioară a timpului în care pot fi trimise mesaje , inclusiv timpul necesar generării mesajului şi timpul necesar transmiterii mesajului în rețea. Procesele pot organiza transmiterea de mesaje în runde, astfel dacă un mesaj nu a ajuns la finele unei runde acesta nu va mai ajunge la receptor.

Informații suplimentare ceasuri vectoriale https://en.wikipedia.org/wiki/Vector_clock

Algoritmul 1. Birman-Schiper-Stephenson

Scop : menține ordinea la trimiterea mesajelor. De exemplu, dacă *send*(m_1) -> *send*(m_2), atunci pentru toate procesele ce primesc m_1 şi m_2, este adevarată relația *receive*(m_1) -> *receive*(m_2). Astfel , m_2 nu va fi trimis procesului receptor decât după mesajul m_1, fiind astfel necesar un bufer pentru mesajele aflate în aşteptare

- Fiecare mesaj are asociat un vector ce conține informație pentru receptor, astfel încât acesta să poată determina dacă un alt mesaj l-a precedat
- Mesajele sunt de tip broadcast
- Ceasurile sunt actualizate la trimiterea mesajelor

Informații suplimentare pot fi obținute la adresa
https://www.cc.gatech.edu/fac/Mustaque.Ahamad/pubs/plausible.ps
respectiv https://github.com/besuikerd/distributedalgorithms/tree/master/BirmanSchiperStephenson

Pseudocodul protocolului (adaptat din http://nob.cs.ucdavis.edu/classes/ecs251-2000-01/distclock.html**)**
Notație

n procese
P_i proces
C_i ceas vectorial asociat procesului P_i; elementul j este $C_i[j]$ și conține pentru P_i ultima valoare pentru timpul curent în procesul P_j
t^m timestamp vectorial pentru mesajul m (după incrementarea ceasului local)

P_i trimite un mesaj către P_j

1. P_i incrementează $C_i[i]$ și setează timestamp $t^m = C_i[i]$ pentru mesajul m.

P_j recepționează un mesaj de la P_i

1. Când P_j, j != i, recepționează m cu timestampul t^m, va întârizia livrarea mesajului până când ambele condiții sunt îndeplinite:
   a. $C_j[i] = t^m[i] - 1$; și
   b. pentru $k <= n$ și k != i, $C_j[k] <= t^m[k]$.
2. La livrarea mesajului către P_j, se actualizează ceasul vectorial al lui P_j'.
3. Verifică mesajele buferate pentru a vedea dacă sunt mesaje ce pot fi livrate.

Algoritmul 2. Schiper-Eggli-Sandoz Protocol

Scop:Asigură că mesajele vor ajunge la procesul receptor în ordinea trimiterii. Acest protocol nu va folosi mesaje broadcast. Fiecare mesaj are asociat un vector ce conține informație pentru fiecare receptor astfel încât acesta să determine correct dacă un al mesaj l-a precedat.ceasurile se actualizează doar la trimiterea mesajelor.
Informații suplimentare pot fi obținute la adresa https://link.springer.com/chapter/10.1007/3-540-51687-5_45 respectiv, https://github.com/dzzh/in4150/tree/master/1

Pseudocodul protocolului (adaptat din http://nob.cs.ucdavis.edu/classes/ecs251-2000-01/distclock.html)
Notație

n procese
P_i proces
C_i ceas vectorial asociat procesului P_i; elementul j este $C_i[j]$ și contine în P_i ultima valoare pentru timpul curent în procesul P_k
t^m timestamp vectorial pentru mesajul m (stampă după incrementarea ceasului local)
t^i timp curent la procesul P_i
V_i vector pentru procesul P_i de forma $V_i[j] = t^m$, unde P_j este procesul destinație și t^m stampa de timp vectorială; $V_i[j][k]$ este componenta k din $V_i[j]$. V^m vectorul corespunzător mesajului m

P_i trimite un mesaj către P_j

1. Pi trimite message m, timestamp t^m, și V_i, procesului P_j
2. Pi setează $V_i[j] = t^m$

P_j recepționează un mesaj de la P_i

1. Când P_j, j != i, primește m, va întârzia livrarda mesajului dacă ambele condiții sunt îndeplinite:
   a. $V^m[j]$ este setat; și
   b. $V^m[j] < t^i$

2. Atunci când mesajul este livrat către P_j, se vor actualize elementele vectorului V_j cu elementele corespondente din V^m, mai puțin $V_j[j]$, după cum urmează:
   a. dacă $V_j[k]$ și $V^m[k]$ sunt neinițializate , nimic.
   b. dacă $V_j[k]$ este neinițializat și $V^m[k]$ is inițializat, setează $V_j[k] = V^m[k]$.
   c. dacă ambele $V_j[k]$ și $V^m[k]$ sunt inițializate, setează $V_j[k][k'] = \max(V_j[k][k'], V^m[k][k'])$ pentru $k' = 1, ..., n$
3. Actualizează ceasul vectorial al lui P_j's .
4. Verifică mesajele buferate pentru a vedea dacă pot fi livrate

8.2.5.Algoritmi pentru alegere leader

Diverse procesări în context distribuit necesită un anumit proces cu rol distinctiv (lider sau coordonator) determinat din cadrul unui set de procese, astfel încât procesarea să poată fi coordonată

de acesta. În prezența eșecului procesului ales drept lider, este necesară determinarea unui nou lider pentru a permite continuarea procesării. Pentru cazul în care toate procesele au un număr de identificare unic, alegerea liderului poate fi redusă la operaţia de găsire a unui proces neeşuat, ce deţine cel mai mare identificator.

Informații suplimentare conceptualizare algoritmi :https://en.wikipedia.org/wiki/Leader_election
Exemple de implementare :https://docs.microsoft.com/en-us/azure/architecture/patterns/leader-election

Orice algoritm corect de determinare a acestui proces trebuie să îndeplinească următoarele proprietăţi:

1. Siguranţă: un proces, fie nu cunoaşte coordonatorul, fie cunoaşte identificatorul procesului cu cel mai mare număr de identificare.
2. Longevitate: în cele din urmă, un proces este supus eşecului sau îl cunoaşte pe coordinator.

Unul din cei mai cunoscuți algoritmi de alegere lider a fost propus de Garcia-Molina şi utilizează trei tipuri de mesaje:

1. Mesaje de tipul Election (alegere) – anunţă alegerea
2. Mesaje de tipul Answer (răspuns) – procesele răspund la o alegere
3. Mesaje de tipul Leader (coordonator) – coordonatorii aleşi se anunţă în sistem

Algoritmul Bully. Modul de execuțiea acestui algoritm este descries în continuare.Un proces începe o alegere în momentul în care realizează, prin intermediul unui timeout, faptul că recentul coordonator a eşuat, sau dacă primeşte un mesaj de tipul Election. Când se începe procesul de alegere, un proces trimite mesajul Election la toate procesele cu un număr de identificare mai mare. Dacă procesul nu primeşte niciun mesaj de tipul Answer într-un interval de timp predefinit, procesul care a început alegerea va decide că trebuie să devină coordonator şi va trimite un mesaj de tipul Coordinator către toate celelalte procese.

În cazul în care primeşte un mesaj de tipul Answer, procesul care a iniţiat alegerea va aştepta un interval de timp predefinit pentru un mesaj de tipul Coordinator. Un proces care primeşte un mesaj de tipul Election poate anunţa instantaneu că el este coordonatorul dacă ştie sigur că deţine numărul de identificare cel mai mare. Altfel, va începe şi el o alegere subordonată prin trimiterea unui mesaj de tipul Election către procesele care au numerele de identificare mai mari. Algoritmul este numit şi algoritm bully (forţa brută) deoarece procesul cu numărul cel mai mare de identificare va fi mereu coordonator.

Descriere nonformală

Algoritm Bully (Garcia-Molina ,1982, adaptare din http://www.cs.vu.nl/~tcs/da/daslides.pdf)

1. Fiecare proces are un identificator unic (numărul procesului)
2. Fiecare proces cunoaşte identificatorii tuturor celorlalte procese
3. Procesele pot fi active sau nu
4. Un proces este **lider**
    a)Liderul îndeplinește un rol special în sistemul distribuit
5. Dacă liderul cade, celelalte procese trebuie să aleagă unul dintre ele pentru a fi noul lider
6. Transferarea mesajelor este sincronă
7. Când un proces observă că liderul este căzut, procesul trimite un mesaj "election" la toate procesele cu identificatori mai mari decât cel propriu
8. Când un proces primește un mesaj "election" de la un proces cu număr mai mic, procesul trimite un mesaj "OK" înapoi la procesul cu număr mai mic, apoi trimite un mesaj "election" tuturor proceselor cu numere mai mari
9. Când un proces primește un mesaj "OK", procesul nu mai face nimic şi așteaptă un mesaj "leader"
10. Dacă un proces nu primește un răspuns la un mesaj "election", acest proces devine noul lider
11. Noul lider trimite un mesaj de "lider" tuturor proceselor, informându-i cine este noul lider(coordonatorul)
12. Procesul cu numărul cel mai mare devine întotdeauna liderul (bully process)
13. Când începe un proces, inițiază o alegere și devine lider (sau nu) dacă este cel mai mare număr

Algoritmul Ring. O alternativă la algoritmul mai sus prezentat este utilizarea unui algoritm cu organizare topologică a proceselor de tip inel. În această abordare, toate procesele sunt ordonate sub forma unui inel logic şi fiecare proces cunoaşte informaţii despre structura inelului. Mesajele implicate sunt doar de două tipuri – mesaje de tipul *Election* şi de tipul *Lider*. Un proces începe o alegere în momentul în care realizează că precedentul coordonator a eşuat. O nouă alegere porneşte prin trimiterea unui mesaj de tipul Election la primul vecin de pe inel. Mesajul conţine identificatorul procesului, acesta este transmis prin inel, fiecare proces adăugându-şi propriul identificator. Alegerea este completă când mesajul ajunge înapoi la procesul iniţiator. Bazat pe conţinutul mesajului, procesul respectiv determină numărul de identificare maxim şi trimite un mesaj de tipul *Lider*, specificând faptul că procesul respectiv este câştigătorul.

Algoritm Ring (Chang Roberts 1979, adaptare din http://www.cs.vu.nl/~tcs/da/daslides.pdf)

1. Procesele sunt aranjate într-un inel în ordinea crescătoare a numărului procesului
2. Când un proces constată că liderul este căzut, procesul trimite un mesaj "election" la următorul proces din ring
    a. Mesajul "election" conține o listă cu toate numerele de proces care au văzut până acum mesajul "election"
    b. Inițial, lista conține numărul procesului care a inițiat alegerile
3. Procesul așteaptă să primească o confirmare de la destinatar
4. Dacă nu există nicio confirmare, procesul trimite mesajul "election" la următorul proces în ring și așa mai departe până când primește o confirmare
5. Când un proces primește un mesaj "election", procesul se adaugă la lista proceselor din mesaj și transmite mesajul mai departe
6. În cele din urmă procesul care a început alegerile primește un mesaj "election" care a parcurs întreg ringul
    a. Lista din mesaj conține toate procesele care sunt încă în desfășurare
    b. Procesul cel mai mare număr va fi liderul
    c. Procesul schimbă mesajul într-un mesaj "leader" și îl transmite în jurul inelului
7. Când un proces primește un mesaj "leader", procesul înregistrează cine este noul lider și transmite mesajul
8. În cele din urmă, procesul care a început alegerile primește un mesaj "lider" care a parcurs întreg ringul
    a. Procesul nu mai transmite mesajul și alegerile se termină

8.2.6.Excluderea mutuală distribuită

Unele dintre problemele întâmpinate în domeniul concurenţei din cadrul sistemelor distribuite sunt similare cu cele întâlnite în studiul sistemelor de operare şi a aplicaţiilor multithread. În particular, ne referim aici la problemele impuse de condiţiile de concurenţă care apar în momentul în care procese multiple accesează resurse comune. În sistemele nedistribuite, aceste probleme sunt rezolvate prin implementarea excluderii mutuale, implicând folosirea primitivelor sistemului de operare (primitive locale) cum ar fi lacătele, semafoarele şi monitoarele. În sistemele distribuite, rezolvarea problemelor datorate concurenţei devine mult mai complexă, datorită lipsei de resurse comune directe (cum ar fi memoria, regiştrii CPU, etc.), datorită lipsei unui ceas global, a unei stări globale unice şi datorită prezenţei întârzierilor de comunicare.

Atunci când se dorește accesul concurent la o resursă dintr-un sistem distribuit, e necesară existenţa unor mecanisme care ajută la prevenirea condiţiilor de concurenţă în timp ce procesele se află în cadrul unor secţiuni critice. Aceste mecanisme trebuie să asigure existenţa următoarelor trei proprietăţi: a)siguranţa: cel mult un proces are dreptul de a executa secţiunea critică la un moment dat. b) longevitatea: la un moment dat, cererile de a intra şi de a ieşi din secţiunea critică trebuie să fie efectuate cu succes și c) ordonarea: cererile sunt procesate într-o ordine de tipul happened-before.

Au fost propuse mai multe soluții pentru rezolvarea problemei accesului concurrent la resurse în cazul proceselor distribuite și anume : metoda unui Server central, metoda Token Ring sau algoritmi

specializaţi ce utilizează ceasurile logice pentru a determina ordonarea şi a asigura serializarea accesului la resursă (algoritmii Ricart-Agrawala şi Maekawa), soluții ce vor fi descrise în continuare.

Ipotezele generale valabile pentru toți aceşti algoritmi sunt următoarele :nu există memorie partajată, nu este utilizabil un ceas global, fiecare proces are propriul ceas şi procesele comunică prin schimb sigur de mesaje.

A) Algoritmi de excludere mutuală distribuită

Algoritm 1: Server Central. Cea mai simplă abordare este de a utiliza un server central care controlează intrarea şi ieşirea din secţiunile critice. Procesele trebuie să trimită cereri de intrare şi ieşire dintr-o secţiune critică către un server coordonator. Acesta acordă permisiuni de intrare prin trimiterea unui *token* spre procesul care a înaintat cererea. La părăsirea secţiunii critice, token-ul este returnat serverului. Procesele care doresc să intre într-o secţiune critică în timp ce un alt proces deţine token-ul, sunt aşezate într-o coadă. Când token-ul este returnat, procesul de la începutul cozii primeşte token-ul şi totodată permisiunea de a intra în secţiunea critică.Această schemă este uşor de implementat, dar nu prezintă o scalabilitate bună, datorită existenţei unei autorităţi centrale , mai mult, este vulnerabilă din punctul de vedere al eşuării serverului central.

Algoritm 2: Token Ring (inel cu jeton) .O topologie de procese mai sofisticată este cea care organizează toate procesele într-o structură logică de inel, alături de un mesaj de tipul token care parcurge inelul continuu. Înainte de a intra în secţiunea critică, un proces trebuie să aştepte până când token-ul trece prin dreptul său. În acel moment, procesul reţine token-ul până în momentul părăsirii secţiunii critice. Dezavantajul acestei abordări este faptul că inelul implică o întârziere de N/2 salturi, lucru care limitează scalabilitatea din nou. Mai mult, mesajele de tipul token consumă lăţime de bandă şi nodurile sau canalele eşuate pot rupe inelul. O altă problemă este reprezentată de pierderea token-ului – fapt cauzat de prezenţa eşecurilor. În plus, dacă un proces nou se alătură reţelei sau doreşte să o părăsească, este nevoie de logică adiţională de management pentru integrare.

Comparație și analiza algoritmilor

Pentru compararea algoritmilor de excludere mutuală distribuită este necesară analiza numărului de mesaje inter-schimbate per intrare/ieşire dintr-o secţiune critică, întârzierea unui proces înainte ca acesta că primească permisiunea de a intra în secţiunea critică, respectiv nivelul de siguranță al algoritmilor (ce probleme posibile întâmpină aceştia în execuție).

Algoritmul centralizat are nevoie de un total de 3 mesaje inter-schimbate la fiecare execuţie a unei secţiuni critice (două pentru intrare şi unul pentru ieşire). După ce un proces trimite cererea pentru obţinerea permisiunii de a intra într-o secţiune critică, acesta trebuie să aştepe pentru un timp minim în care are loc inter-schimbarea a două mesaje (unul pentru deţinătorul curent - ca să returneze token-ul coordonatorului, şi unul pentru coordinator – ca să trimită token-ul la procesul aflat în aşteptare). Cea mai mare problemă pe care o are acest algoritm referă situația în care coordonatorul eşuează (sau devine nedisponibil) şi prin urmare, tot algoritmul eşuează.

Pentru algoritmul inel, numărul mesajelor inter-schimbate per intrare şi ieşire din secţiunea critică, depinde de cât de des au nevoie procesele să intre în secţiune. Cu cât mai rar doresc procesele să intre, cu atât mai mult timp va dura drumul token-ului prin inel şi cu atât mai mari vor fi costurile ,în termenii mesajelor inter-schimbate de intare într-o secţiune critică. Din punctul de vedere al întârzierilor depinzând de locaţia token-ului, va exista un număr de 0 până la n-1 mesaje înainte ca un proces să intre în secţiune. Cele mai mari probleme întâmpinate de acest algoritm sunt pierderea tokenului şi ruperea inelului de către un proces eşuat. Este posibilă, desigur, prevenirea acestora, prin oferirea tuturor proceselor de informaţii referitoare la structura inelului. În acest mod, nodurile eşuate pot fi evitate.

B)Algoritmi de excludere mutuală ce folosesc ceasuri logice

Algoritm 1. Algoritmul Ricart-Agrawala

Ricart şi Agrawala [Ricart- Agrawala 1981] au propus un algoritm pentru excluderea mutuală distribuită, care utilizează ceasuri logice. fiecare proces participant p_i reţine un ceas Lamport şi toate procesele trebuie să fie capabile să comunice în perechi. În orice moment, un proces se poate afla doar în una din următoarele trei stări:

1. *Released*: În afara secţiunii critice
2. *Wanted*: Aşteptând să intre în secţiunea critică
3. *Held:* În interiorul secţiunii critice

Dacă un proces doreşte să intre în secţiunea critică, realizează operaţia de multicast a unui mesaj L_i,p_ii şi aşteaptă până la primirea unui răspuns de la oricare alt proces. Procesele operează astfel:

- Dacă un proces este în starea *Released,* va răspunde instantaneu la oricare cerere de intrare în secţiunea critică.
- Dacă procesul este în starea *Held,* va întârzia răspunsul la oricare cerere până în momentul ieşirii din secţiunea critică.
- Dacă procesul este în starea *Wanted*, va răspunde la oricare cerere instantaneu doar dacă timestamp-ul cererii este mai mic decât cel din cadrul propriei sale cereri.

Caracteristicile algoritmului sunt:

- Imbunătățirea algoritmului original propus de Lamport
- Control distribuit
- Necesită o metodă de asignare a timpilor către evenimente
- Nu există ceas global
- folosește „ceasul Lamport”
- Fiecare proces/procesor își păstreaza propriul timp
- Evenimentele locale sunt asignate în ordinea strictă a creșterii marcajelor de timp
- Fiecare mesaj dintre procese e însoțit de un marcaj de timp care indică timpul procesului care a trimis mesajul
- Când un mesaj este recepționat, marcajul său de timp este comparat cu timpul local
- Daca timestamp > timp local, setează timp local = timestamp + 1

Acest aspect impune o relație „s-a intamplat inainte” (happens before) între evenimente, cu proprietățile:

- Dacă evenimentA se întâmplă înaintea evenimentB pe o singură maşină, T(evenimentA) < T(evenimentB)
- Dacă evenimentA este expeditorul mesajului şi evenimentB este destinatarul aceluiaşi mesaj, atunci T(evenimentA) < T(evenimentB)

Această ordonare nu e o ordonare totală, dar poate fi transformată într-una daca se combină timpul cu id-ul unui proces în care un eveniment are loc.

Pseudocodul algoritmului

Algoritmul Ricart-Agrawala este o îmbunătățire a unui algoritm publicat inițial de Lamport, care utilizează ceasul Lamport. Algoritmul are trei componente, care dictează cum se comportă un proces când vrea să intre într-o zonă critică (abreviat CS), când vrea să iasă dintr-o zonă critică și când primește un mesaj de solicitare de la un alt proces. Fiecare proces participant în algoritm reține o coadă cu cereri în așteptare.

Informații suplimentare :
https://en.wikipedia.org/wiki/Ricart%E2%80%93Agrawala_algorithm
https://github.com/vineetdhanawat/ricart-agrawala
https://github.com/madhav5589/Ricart-Agrawala-algorithm-for-distributed-mutual-exclusion

Pseudocodul algoritmului (sursa [2]):

```
enterCS:

construct a request –to- enter message;
assign the current logical time to the request;
send the message to each other process;
wait for okay response from each other process;

receiveRequestmessage;
if this process is in the CS
enqueue the request;

else if this process is not waiting to enter the critical section
send okay to requestor;

else//this process is waiting to enter the critical section

if (this.request.timeStamp <incomingRequest.timeStamp)
enqueue the request;
else
       send okay to requestor;

exitCS:

while(request queue not empty){
dequeue a request message;
send okay to requestor;}
```

Analiza algoritmului. Două procese nu pot avea același marcaj de timp si fiecare proces trebuie să fie de acord cu ordonarea cererilor. Algoritmul eșuează dacă oricare din procesele participante este incapabil să răspundă la mesaje. Se poate observa faptul că 2(n-1) mesaje sunt necesare pentru fiecare intrare în zona critică. O posibilă îmbunătățire ar fi implementarea unui proces care să confirme fiecare mesaj de solicitare cu OK sau REJECT. Dacă un proces eșuează, celalalt proces va putea detecta aceasa şi ori va elimina mesajul din grup, ori va opri aplicația. Variante mult mai scalabile ale acestui algoritm cer fiecărui proces individual să contacteze subseturi ale vecinilor în momentul în care se dorește intrarea în secţiunea critică. Din păcate, eșecul oricărei componente -proces poate afecta intrarea oricăror altor procese în secţiunea critică deci poate periclita execuția algoritmului.

Algoritm 2 .Algoritmul Maekawa

Pentru fiecare proces se definește o mutțime ce contine cereri, notata Ri. Mulțimile de cereri trebuie sa aibă proprietățile:

- Intersectia lui Ri cu Rj nu trebuie să fie vidă
- Fiecare proces aparține mulțimii sale de cereri
- Fiecare mulțime de cereri are același număr de elemente K
- Fiecare proces aparține la exact K mulțimi de cereri

Maekawa a arătat că este posibilă construirea unei mulțimi astfel încât K = O(radical(N)). Algoritmul foloseste trei tipuri de mesaje: *Request, ok, End*. Un proces care solicită intrarea într-o zonă critică trebuie să primeasca OK-uri doar de la membri mulțimii sale de solicitanți.

Informații suplimentare :
https://en.wikipedia.org/wiki/Maekawa's_algorithm

https://github.com/saransh2405/Maekawa-algorithm,https://github.com/sanketc/MaekawaAlgorithm

Pseudocodul algoritmului (sursa [2]):

```
Enter CS:

Send request message to each member of my request set.
Wait for okay messages from each member of my request set.

Exit CS:

Send done message to ecah member of my request set.
receiveRequestMessage
( each process has a granted variable, initialized to false)
 If granted
enqueue the request;
else {granted =true;
   send okay to the requestor;}
 receiveDoneMessage:
if (queue is empty)
granted=false ;else{
   dequeue the request with the earliest timestamp;
send okay top the requestor;}
```

Avantajul acestei metode este faptul ca doar 3*radical(N) mesaje sunt necesare pentru intrarea intr-o zona critica.Problema posibilă este aparitie a unui blocaj (deadlock) la nivelul proceselor, o posibilă rezolvare ar fi următoarea :

- Daca un proces primește un mesaj de *Request* cu un marcaj de timp mai devreme decât timpul curent al propriului mesaj de *ok*, el trimite un mesaj de *interogare* către procesul la care a trimis mesajul de *ok*
- Dacă respectivul proces încă este în starea de așteptare pentru a intra într-o zonă critică, el trimite înapoi un mesaj de *Release*
- Procesul initial poate pune apoi solicitarea primită înapoi în coada de așteptare și trimite un mesaj de *ok* către solicitarea pe care tocmai a primit-o

Comparație și analiza algoritmilor

Algoritmul original al lui Lamport presupune ca fiecare proces sa solicite acces de la celelalte procese, folosind mesaje de 3 tipuri: request, reply, release. Aceasta determină ca pentru fiecare solicitare către zona critica (critical section, presurtat CS), sa fie nevoie de 3*(N-1) mesaje, unde N este numarul de noduri/procese.

In comparatie cu soluția originală Lamport, algoritmul Ricart-Agrawala vine cu o reducere de mesaje transmise la 2*(N-1), deoarece foloseste mesaje doar de tip request și reply. Mai mult de atât, algoritmul lui Maekawa scade numărul de mesaje la 3*radical(N), deoarece el are 3 tipuri de mesaje (request, locked, release), dar ele sunt transmise numai in cadrul grupului de care aparține un anumit proces.

Un aspect diferit pentru cei doi algoritmi este coada care va memora mesajele sosite. Pentru algorimtul Ricart-Agrawala, coada este non-fifo (non first in, first out; primul intrat, primul iesit), fapt care se implementează printr-o coadă simplă care plasează mesajele în ordine aleatoare. Pe de altă parte, algoritmul Maekawa, foloseste o coada fifo uzuală.

8.3.Exemple

8.3.1.Se vor testa exemplele ce ilustrează modul de utilizare a tehnologiei RMI, exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/08_AD.

8.4.Întrebări teoretice

8.4.1. Ce este un ceas logic? Dar un ceas vectorial? Imaginați un protocol care să mențină ceasurile a trei hosturi distribuite în trei continente sincronizate.Ce structuri d edate va avea nevoie un astfel de protocol?

8.4.2.Ce semnificație are conceptul de cauzalitate? Cum poate fi implementat într-un sistem distribuit?

8.4.3. Analizați comparativ din perspectivă funcțională dar și a complexității (respectiv performanței) algoritmii bully și ring.Identificați pentru fiecare avantaje, dezavantaje, posibile optimizări și număr de mesaje schimbate.

8.4.4. Comparați funcțional algoritmii Ricart-Agrawala și Maekawa, analizând complexitatea, dezavantaje și posibile optimizări.

8.5.Probleme propuse

8.5.1.Implementați algoritmii de alegere lider (Bully și Ring) realizați și prezentați o analiză a complexității lor (mesaje schimbate, timp de execuție, scalabilitate). Identificați posibile scenarii de aplicabilitate a fiecărui algoritm.

8.5.2.Implementați algoritmii de excludere mutuală distribuită clasici (Server central și Token Ring) prezentați și realizați o analiză a complexității lor (mesaje schimbate, timp de execuție , scalabilitate).Identificați posibile scenarii de aplicabilitate a fiecărui algoritm.

8.5.3.Implementați algoritmii de excludere mutuală bazați pe ceasuri logice (Ricart Agrawala și Maekawa) prezentați și realizați o analiză a complexității lor. Identificați scenarii adecvate de aplicabilitate a fiecărui algoritm.

8.6. Referințe bibliografice

1.W. Fokkink – Distributed algorithms notes : https://www.cs.vu.nl/~tcs/da/daslides.pdf
2. A Kshemkalyani – Distributed Computing-principles, algorithms and systems
https://eclass.uoa.gr/modules/document/file.php/D245/2015/DistrComp.pdf

9. JMS și JGroups – tehnologii pentru comunicație de grup

Tehnologiile middleware bazate pe cozi sau pe principiul publică-subscrie pot fi folosite pentru proiectarea multor modele de comunicar, fiind relative simple eficiente şi oferind un grad ridicat de performanță și fiabilitate. Apar însă o serie de probleme atunci când trebuie integrate aplicații care folosesc tipuri de mesaje diferite, acest lucru se întâmplă frecvent atunci când se dorește construirea de sisteme software complexe prin interconectarea unor sisteme software mai vechi. Pentru rezolvarea problemelor care apar la integrarea sistemelor software complexe se poate folosi tehnologia middleware cunoscută sub numele Message Oriented Middleware (MOM sau broker de mesaje).

9.1.Obiective

- Studiul claselor și interfețelor specifice sistemului de mesagerie Java - JMS
- Studiul mecanismelor necesare comunicației de grup (toolkitul JGroup)
- Implementarea unor aplicații distribuite folosind mecanismele oferite de JMS, respectiv JGroups

9.2. Concepte

Middleware orientat mesaj (*Message Oriented Middleware* - MOM) oferă abilitatea de procesare a cererilor de transmitere de mesaje în mod asincron, dându-le astfel posibilitatea proiectanţilor şi dezvoltatorilor de a furniza soluţii care să reducă sau să chiar să elimine limitări în transferul de date între componentele unui sistem informatic, crescând în acest fel productivitatea utilizatorului final şi posibilitatea sistemului, în ansamblul lui, de a face faţă unui volum de date de prelucrat care poate creşte semnificativ de-a lungul perioadei de exploatare. Avantajele oferite de MOM sunt următoarele:

- posibilitatea integrării de componente eterogene, rulând pe platforme diferite, într-un sistem informatic distribuit;
- decuplarea aplicaţiilor, prin schimbul asincron de mesaje ce oferă fiecărei componente posibilitatea de a prelucra datele în ritmul propriu, fără a afecta performanţele altor componente ale sistemului;
- creşterea capacităţii sistemului de a face faţă unor volume mari de date, prin posibilitatea de a adăuga dinamic componente pentru a prelucra concurenţial cozi de mesaje omogene;
- posibilitatea proiectării unor sisteme cu arhitecturi flexibile şi agile, care se pot uşor adapta nevoilor concrete de exploatare;

Unul dintre conceptele de bază ale unui *middleware* orientat pe mesaje este acela de livrare asincronă a mesajelor între sistemele conectate într-o reţea. Livrarea asincronă a mesajelor presupune că aplicaţia care trimite un mesaj nu trebuie să aştepte ca respectivul mesaj să fie recepţionat sau prelucrat de aplicaţia căreia i-a fost destinat; aplicaţia care produce mesajul are libertatea de a trimite mesajul în cauză şi continua apoi propriul flux de prelucrare. Mesajele asincrone sunt manipulate ca unităţi de sine stătătoare, în sensul în care fiecare mesaj conţine toate datele de business şi cele de stare necesare pentru a putea fi prelucrat din perspectiva logicii aplicaţiei ce se găseşte la destinaţie. În filosofia de mesagerie asincronă, aplicaţiile utilizează o interfaţă (*Application Programing Interface* - API) simplă pentru a construi mesaje, iar aceste mesaje sunt ulterior pasate către platforma de comunicaţie orientată pe mesaje, pentru ca aceasta din urmă să le livreze către destinaţiile avute în

vedere.Arhitecturile utilizate pentru implementarea MOM pot urmări diferite abordări. De la o arhitectură centralizată, care se bazează pe un server ce realizează rutarea mesajelor, până la arhitecturi descentralizate, în care se distribuie componenta de rutare a mesajelor pe maşinile clienţilor. De asemenea, protocoalele de comunicaţie utilizate la nivelul de reţea transport includ TCP/IP, HTTP, SSL şi IP multicasting.

Este important de menţionat faptul că, în contextul utilizării unei platforme de comunicaţie orientate pe mesaje, termenul de client capătă o semnificaţie mai largă, în sensul în care toate aplicaţiile conectate la platforma de comunicaţie devin clienţi ai acestui *middleware*, chiar dacă din perspectiva logicii funcţionale a sistemului , privit în ansamblul său, pot juca rol de client, respectiv de server. Sistemele de comunicaţie orientate pe mesaje sunt compuse din mesajele aplicaţiilor client ale platformei şi componenta de server de *middleware* pentru rutarea acestor mesaje. Clienţii transmit mesaje către server de mesagerie, mesaje care sunt apoi distribuite de acesta către alţi clienţi ai sistemului. Sistemele de mesagerie pot fi centralizate, descentralizate sau hibride.

Sistemele de mesagerie care se bazează pe o *arhitectură centralizată* folosesc un aşanumit server de mesaje. Serverul de mesaje, care poate fi de asemenea denumit şi *router* de mesaje, sau *broker* de mesaje este responsabil cu livrarea mesajelor între clienţii sistemului de mesagerie. Serverul de mesaje realizează, în acest fel, decuplarea clientului care produce mesajul de cel care consumă respectivul mesaj. Clienţii intră în contact numai cu serverul de mesaje şi nu cu ceilalţi clienţi ai sistemului, permiţând eliminarea sau adăugarea de noi clienţi, fără afectarea sistemului ca un întreg. În mod normal, o arhitectură centralizată utilizează o topologie de tip stea. În configuraţia cea mai simplă, avem de aface cu un broker de mesaje plasat central, la care se conectează toţi clienţii sistemului. Arhitectura de tip stea oferă un număr minim de conexiuni de reţea, interconectând totodată fiecare componetă a sistemului cu toate celelalte.În practică, serverul central de mesagerie poate fi de fapt un cluster de servere distribuite fizic, dar operând ca o singură entitate la nivel logic. Plecând de la această observaţie, o arhitectură MOM centralizată poate furniza un model generic de mesagerie pentru sisteme extreme de complexe, oferind un grad înalt de flexibilitate şi posibilităţi facile de redimensionare.

A*rhitecturile descentralizate* folosesc la nivel de reţea protocolul *IP multicast*. Un sistem de mesagerie bazat pe multicasting nu are un server central pentru livrarea mesajelor. Unele dintre funcţionalităţile care ar fi oferite de server sunt încorporate în partea de client (persistare, facilităţi de lucru cu tranzacţii, aspecte legate de siguranţa comunicării), în timp ce rutarea mesajelor este delegată către nivelul de reţea, prin folosirea protocolului *IP multicast*. Acest protocol oferă posibilitatea aplicaţiilor de subscrie la unul sau la mai multe grupuri de *multicast*. Fiecare grup de *multicast* foloseşte o adresă IP de reţea prin care fiecare mesaj ce este primit va fi redistribuit către toţi membrii grupului. În acest fel, aplicaţiile pot trimite mesaje către o adresă de IP de tip *multicast* şi se aşteaptă ca nivelul de reţea să asigure distribuirea acestor mesaje în mod corespunzător membrilor care au subscris la acea adresă IP. Rutarea de realizează în mod automat, la nivel de reţea, fără a mai fi necesar un server dedicat. Cu toate acestea, aşa cum am menţionat iniţial, anumite funcţionalităţi oferite de server vor fi necesar a fi incluse în fiecare client, aspect ce poate conduce la încărcarea clienţilor şi la distribuirea de funcţionalităţi care ar fi mai simplu de gestionat într-un singur loc, cel puţin la nivel logic.

9.2.2.Modele de mesagerie furnizate de *Java Message Service* (JMS)

Cu toate că interfaţa pusă la dispoziţie de *Java Message Service* (JMS) nu s-a modificat semnificativ de la introducerea ei în 1999, maniera în care soluţiile *middleware* orientat pe mesaje sunt utilizate în sisteme şi aplicaţii diverse s-a schimbat semnificativ. Sistemele de mesagerie sunt astăzi utilizate pentru a rezolva probleme legate de siguranţa livrării datelor între aplicaţii, respectiv probleme legate de capacitatea de adaptare dinamică în funcţie de volumul datelor de prelucrat.

Java Message Service (JMS) oferă două tipuri de modele de mesagerie:

- punct la punct (*point-to-point*, sau prescurtat *p2p*) ;
- publicare şi subscriere (*publish-and-subscribe*/*pub-sub).*

Principial, modelul *pub/sub* presupune publicarea mesajelor de la unul către mai mulţi (*broadcast*), în timp ce modelul *p2p* este dedicat trasmiterii de mesaje unu-la-unu (Figura 9.1).

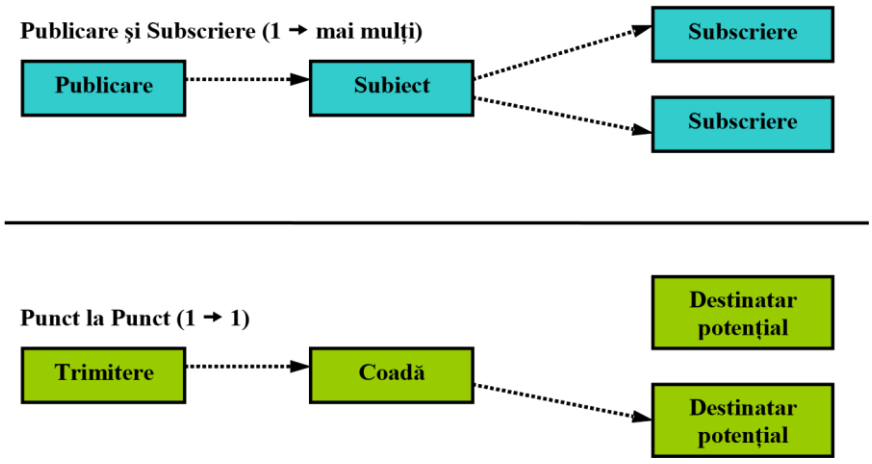


Figura 9.1.Modelele de mesagerie furnizate de JMS API

Din perspectiva JMS, clienţii conectaţi la platforma de mesagerie se numesc *clienţi* JMS, iar sistemul de mesagerie, cel care furnizează serviciul de distribuire a mesajelor între clienţi, se numeşte JMS *provider*. În general, o aplicaţie JMS este un sistem informatic care este compus din mai mulţi clienţi JMS şi un server JMS. În acest context, un client JMS care trimite un mesaj în sistem se numeşte producător al acelui mesaj, în timp ce un client JMS care primeşte un mesaj din sistem se numeşte consumator al respectivului mesaj. Trebuie remarcat faptul că un client JMS poate fi în acelaşi timp şi producător şi consumator de mesaje.

a)Modelul de comunicare punct-la-punct (p2p)

Acest model permite clienţilor JMS să trimită şi să primească mesaje, atât în manieră sincronă cât şi în manieră asincronă, prin intermediul unui canal de comunicaţie care urmează filosofia de acces a unei cozi. În modelul *p2p* producătorii de mesaje se numesc *senders*, iar consumatorii de mesaje se numesc *receivers*. În mod tradiţional, modelul *p2p* este un model în care mesajele sunt în mod activ cerute din coadă (*pull-based* sau *poll-based*) de către aplicaţia căreia îi sunt destinate, în loc să fie făcute disponibile către aplicaţie imediat ce sunt recepţionate (*pushed*). Depinzând de implementare, chiar şi în modelul *p2p*, serverul de mesaje poate însă să facă disponibile mesajele din coadă (*push*) aplicaţiei care s-a conectat la aceasta. (de ex. OpenMQ)Una dintre caracteristicile distinctive ale modelului *p2p* este aceea că un mesaj care este trimis către o coadă este primit de către o singură aplicaţie client şi numai una, chiar dacă pot asculta mai multe aplicaţii client la o aceeaşi coadă de mesaje.
Acest model oferă două mecanisme de transmitere a mesajelor la nivel de aplicaţie client:

- trimitere de mesaj fără a aştepta răspuns de confirmare (*fire and forget)*;
- transmisie de tip cerere/răspuns, în manieră asincronă.

b)Modelul de comunicare publică-subscrie

În modelul *pub/sub*, mesajele sunt publicate către un canal virtual de comunicaţie numit *topic*. Aplicaţiile care produc mesajele trimise către acest canal se numesc aplicaţii care publică mesaje, în timp ce aplicaţiile care consumă mesaje de la un anume *topic* se numesc aplicaţii care subscriu la acel *topic*. Spre deosebire de modelul *p2p*, mesajele publicate la un *topic* pot fi primite de mai multe aplicaţii care subscriu la acel *topic*. Această tehnică este uneori numită *broadcast* şi reprezintă modalitatea prin

care un mesaj este distribuit către toate aplicaţiile interesate în recepţionarea lui. Fiecare client al serverului de mesagerie care subscrie la un anumit *topic*, va primi o copie a fiecărui mesaj publicat către acel *topic*. În modelul *pub/sub* mesajele sunt puse imediat la dispoziţia aplicaţiei client (*pushed-based*) care a subscris la un anumit *topic*, fără ca această să ceară în mod explicit furnizarea mesajelor spre prelucrare.

Modelul *pub/sub* oferă un tip de conexiune care nu conduce la o strânsă cuplare a aplicaţiile implicate. Clientul serverului de mesagerie care publică mesaje nu are, în general, cunoştinţă de câţi clienţi care consumă acele mesaje au subscris la *topic* şi ce anume inteţionează aceştia să facă cu aceste mesaje. De exemplu, o aplicaţie care publică date legate de tranzacţiile realizate de către o instituţie bursieră, nu trebuie în mod necesar să ştie câţi clienţi au subscris la aceste date şi felul în care aceşti clienţi prelucrează datele avute în vedere.

Din perspectiva clienţilor care subscriu la un *topic*, modelul *pub/sub* oferă posibilitatea realizării a două tipuri de conexiuni:

- conexiune care nu se realizează într-o manieră durabilă, în sensul că aplicaţia client primeşte mesajele care sunt publicate la subiectul la care a subscris numai pe durata existenţei conexiunii;
- conexiune durabilă, prin care clientul care subscrie la *topic* va primi toate mesajele publicate la acel *topic*, indiferent dacă este conectat sau nu în mod neîntrerupt la serverul de mesagerie; în cazul în care clientul care a subscris de o manieră durabilă la un *topic*, se deconectează de la server sau pierde dintr-un anumit motiv această conexiune, dacă ulterior se reconectează, va primi toate mesajele care au fost publicate la subiectul la care a subscris pe toată durata în care nu a fost conectat la server.

Modelul *p2p*, mai ales în cazul mecanismului de cerere/răspuns asincron, tinde să cupleze mai strâns aplicaţiile implicate, decât modelul *pub/sub*. De exemplu, un trading GUI poate trimite un ordin către o coadă pentru cumpărarea sau vânzarea unui instrument financiar şi apoi să aştepte pentru un răspuns de confirmare, conţinând identificatorul unic în sistem al ordinului, generat se serverul de gestiune a ordinelor care prelucrează mesajele trimise către coada respectivă. Pe de altă parte, modelul *p2p* oferă posibilitatea optimizării încărcării sistemului (*load balancing*) în cazul aplicaţiilor care consumă şi prelucrează mesajele trimise către o coadă. În acest sens se pot lansa în mod dinamic multipli consumatori ai mesajelor unei aceleeaşi cozi, multiple instanţe ale aceleeaşi aplicaţii, putându-se construi o logică de distribuire optimală a mesajelor cozii între aceste instanţe. Trebuie menţionat faptul că specificaţiile oferite de JMS nu includ reguli privid distribuirea mesajelor între mai multe aplicaţii destinatar, însă anumite produse comerciale au ales să implementeze abilităţi privind echilibrarea încărcării consumatorilor. Faţă de modelul *pub/sub*, modelul *p2p* oferă posibilitatea aplicaţiilor client de a consulta coada de mesaje (*queue browser*) şi vizualiza conţinutul mesajelor înainte de a le consuma (extrage din coadă).

9.2.3.Interfeţe de comunicare JMS

JMS este o interfaţă de comunicare orientată pe mesaje care a fost creată de către Sun Microsystems pe baza specificaţiilor JSR-914 (*Java Specification Request*) în 2002. JMS nu este un sistem de mesagerie în sine, ci o interfaţă abstractă împreună cu clasele necesare clienţilor unui sistem de mesagerie pentru ca aceştia să poată comunica cu sistemul. Utilizând interfaţa JMS, o aplicaţie client a unui sistem (server) de mesagerie devine portabilă, din perspectiva serverului concret de mesagerie la care se conectează.

Crearea interfeţei JMS a fost un efort al întregii industrii software. Obiectivul iniţial a fost acela de a furniza un API Java pentru conexiunea la sisteme de mesagerie deja existente la acel moment. Ulterior, scopul iniţial al proiectului început de Sun Microsystems s-a lărgit, mergându-se pe ideea oferirii unei alternative viabile de comunicare, prin intermediul unui *middleware* orientat pe mesaje, la paradigma de comunicare creată prin existenţa sistemelor bazate pe RPC (*Remote Procedure Call*), precum CORBA şi Enterprise JavaBeans.

Interfaţa JMS este formată din trei componete principale:

- partea generală a interfeţei;
- interfaţa *p2p*;
- interfaţa *pub/sub*.

Interfaţa p2p este utilizată exclusiv pentru modelul de mesagerie bazat pe cozi, în timp ce interfaţa *pub/sub* este destinată utilizării exclusive a modelului de mesagerie bazat pe subiecte (*topics*). API-ul general este compus din şapte interfeţe dedicate trimiterii şi recepţionării de mesaje:

ConnectionFactory,Destination,Connection,Session,Message,
MessageProducer,MessageConsumer

Din aceste şapte interfeţe generice (Figura 9.2), conform specificaţiilor JMS, *ConnectionFactory şi Destination* trebuie să fie obţinute de la serverul de mesagerie, utilizând JNDI (*Java Naming and Directory Interface*).

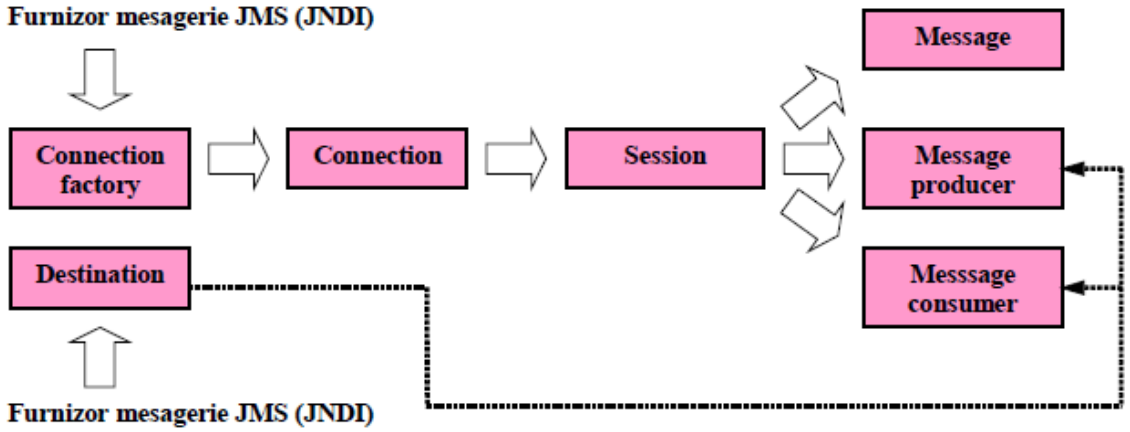


Figura 9.2.Interfeţe generice JMS API

Celelalte interfeţe sunt create prin diversele metode puse la dispoziţie de JMS API. De exemplu, odată ce avem creat *ConnectionFactory*, putem instanţia un obiect de tip *Connection*. Odată ce avem un obiect de tip *Connection,* putem crea unul de tip **Session**, iar mai departe, în cadrul instanţei *Session* putem crea obiecte de tip *Message, MessageProducer şi MessageReceiver*

. În JMS, obiectul *Session* deţine partea tranzacţională a lucrului cu mesaje şi nu obiectul *Connection,* aşa cum este cazul interfeţei JDBC. Aceasta înseamnă că, atunci când utilizează JMS, o aplicaţie va avea în mod normal numai o singură instanţă de tip *Connection*, însă poate deţine o colecţie de obiecte de tip *Session.*

Există, de asemenea, în cadrul JMS API, o multitudine de interfeţe care sunt destinate gestionării excepţiilor, priorităţii mesajelor şi manierei în care acestea sunt persistate.Interfaţa JMS pentru modelul de comunicare *p2p* se referă în mod particular la mesageria bazată pe cozi de mesaje. Interfeţele utilizate pentru trimiterea şi primirea de mesaje în cadrul acestui model sunt următoarele: *QueueConnectionFactory,Queue,QueueConnection*
QueueSession,Message,QueueSender,QueueReceiver.
Întreaga ierarhie de interfeţe şi maniera prin care acestea interacţionează unele cu altele sunt similare modelului general. La nivel sintactic, reflectat până la urmă şi în semantica fiecărei interfeţe, conceptul

generic de destinaţie este înlocuit cu cel de coadă (Figura 9.3.). Aplicaţiile care utilizează modelul de mesagerie *p2p* vor folosi în mod normal interfeţele bazate pe cozi în locul interfeţelor generice

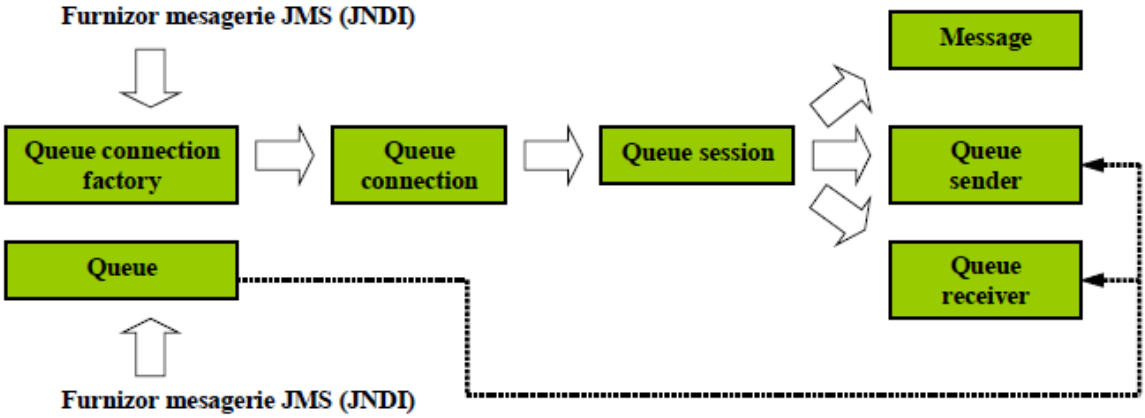


Figura 9.3. Interfețe JMS pentru modelul de comunicare bazat pe cozi de mesaje

Modelul de comunicare pub/sub este oferit de JMS în aceeaşi manieră standardizată de prezentare a interfeţelor. La nivel sintactic cuvântul *Queue* este înlocuit de cuvântul *Topic*. Interfeţele pentrumodelul *pub/sub* sunt următoarele:

TopicConnectionFactory,Topic,TopicConnection
TopicSession,Message,TopicPublisher,TopicSubscriber .

Singurele excepţii, din perpectiva numelui, sunt reprezentate de interfeţele*TopicPublisher* şi *TopicSubscriber*. Însă, aşa cum subliniam mai devreme, JMS API este foarte intuitiv în ceea ce priveşte

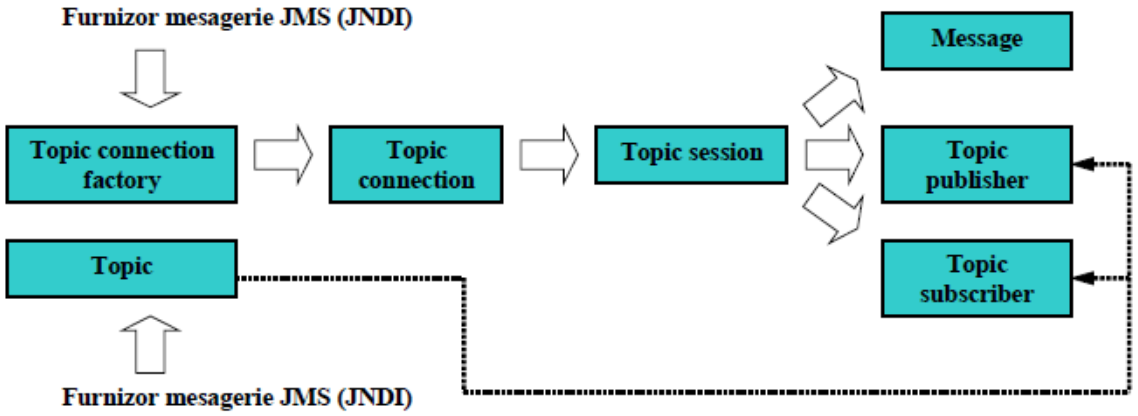


Figura 9.4. Interfețe JMS pentru comunicarea în modelul pub/sub

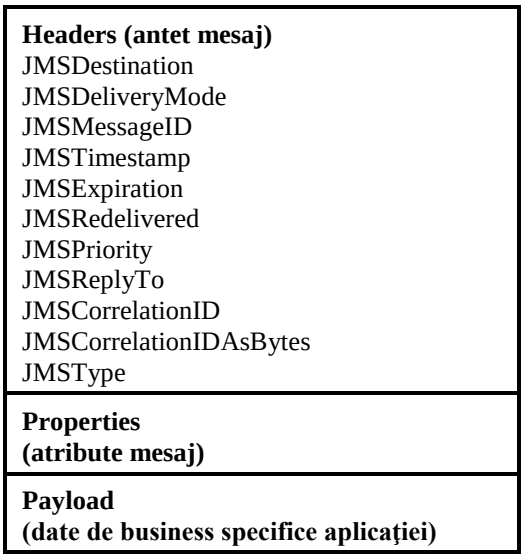
denumirea şi utilizarea interfeţelor pe care le pune la dispoziţie. Din acest punct de vedere, modelul *pub/sub* utilizează *topics* împreună cu *publishers* şi *subscribers*, acolo unde modelul *p2p* foloseşte *queues* împreună cu *senders* şi *receivers*. Aşa cum este reiese şi din Figura 9.4 terminologia şi relaţiile între interfeţele oferite pentru canalizarea mesajelor în cadrul modelului *pub/sub* se mulează fidel pe cele ale API-ul generic oferit de JMS.

9.2.4.Structura unui mesaj JMS

Vom prezentării structura internă a unui mesaj JMS, respectiv a părţilor care compun mesajul:

- componetele de antet (*headers*) ;

- atributele mesajului (*properties*) ;
- diverse tipuri de date utile manipulate de aplicaţie (*message payloads*).



Clasa *Message* reprezintă cea mai importantă parte din specificaţiile care stă la baza JMS. Toate datele şi evenimente utilizate de o aplicaţie fundamentată pe JMS sunt comunicate prin mesaje, celelalte componente ale interfeţei JMS au menirea de facilita transferul acestor mesajelor.Un mesaj JMS joacă un rol unic domeniul sistemelor distribuite. El poate transporta atât date specifice de aplicaţie, cât şi notificări pentru evenimente legate de activitatea aplicaţiilor sistemului. Un mesaj JMS nu este o comandă, prin el se transferă date şi se poate informa aplicaţia care l-a recepţionat că a avut loc un anumit eveniment. Un astfel de mesaj nu îi impune aplicaţiei care îl recepţionează ce anume să facă cu el, iar aplicaţia care l-a produs nu aşteaptă după un răspuns. Se obţine în acest fel o decuplare a producătorului mesajului de consumatorul acestuia, făcând sistemele orientate pe mesaje mult mai dinamice şi mai flexibile decât cele bazate pe paradigma cerere/răspuns realizate în manieră sincronă.Mesajele JMS pot fi de mai multe tipuri, în funcţie de maniera în care sunt organizate datele utile de transferat (*payload*), respectiv pot fi foarte structurate, cum sunt *StreamMessage* şi *BytesMessage*, sau aproape deloc structurate, cum sunt tipurile *TextMessage, ObjectMessage şi MapMessage*. Aşa cum am menţionat mai sus, în toate cazurile, mesajele pot transporta atât notificări cât şi date de business.

Antetul unui mesaj JMS conţine date globale ce descriu cine sau ce aplicaţie a creat mesajul, când a fost acesta creat, cât timp urmează să fie valide datele de business pe care le conţine etc. Antetul conţine, de asemenea, informaţii de rutare a mesajului, care descriu destinaţia acestuia (*topic* sau *queue*), cum trebuie să se realizeze confirmarea de primire a acestuia ş.a.

Fiecare mesaj JMS conţine un set de date standard în antet. Fiecare dintre acestea este accesată printr-un set de metode care urmează o sintaxă similară:

- setJMS< HEADER >()
- getJMS<HEADER>()

Prezentăm în continuare toate aceste metode de acces la componetele antetului unui mesaj JMS, ele făcând parte din descrierea interfeţei *Message:*

Elementele de antet JMS sunt grupate în două categorii: elemente asignate automat şi elemente asignate de programator.

Elementele de antet asignate automat

Majoritatea antetelor mesajelor JMS sunt asignate automat. Valorile elementelor acestor antete sunt setate de serverul de mesagerie atunci când mesajul este livrat, astfel încât valorile asignate de

programator prin utilizarea metodelor *setJMS<HEADER>()* sunt ignorate. Cu toate acestea, unele elemente de antet asignate automat pot fi setate în mod programatic atunci când se instanţiază clasele *Session şi MessageProducer* (respectiv *TopicPublisher).* În această situaţie se găsesc elementele de antet *JMSDeliveryMode şi JMSPriority.*

JMSDestination - Prin acest element de antet se identifică destinaţia mesajului, care poate fi fie un obiect de tip *Topic* fie unul de tip *Queue,* ambele derivate din tipul *Destination*. Identificarea destinaţiei mesajului este importantă, mai ales pentru clienţii care consumă mesaje de la mai mult de un *topic* sau *queue*:

```
Topic destination = (Topic) message.getJMSDestination();
```

JMSDeliveryMode - JMS oferă două moduri de livrarea mesajelor: persistent şi nepersistent. Un mesaj persistent este livrat o dată şi numai o singură dată, ceea ce înseamnă că dacă serverul de mesagerie înregistrează o disfuncţionalitate mesajul nu este pierdut; el va fi livrat după ce serverul îşi reface starea. Un mesaj nepersistent este livrat cel mult o dată, ceea ce implică faptul că el poate fi pierdut pentru totdeauna, dacă serverul de mesaje eşuează în funţionare. În ambele moduri de livrare, persistent şi nepersistent, serverul de mesagerie nu trebuie să trimită un mesaj către un acelaşi consumator mai mult de o dată (lucru posibil totuşi prin utilizarea *JMSRedelivered).*

```
int deliveryMode = message.getJMSDeliveryMode(); if (deliveryMode
== javax.jms.DeliveryMode.PERSISTENT) { ...
} else { // înseamnă DeliveryMode.NON_PERSISTENT ...}
```

Modul de livrare al unui mesaj poate fi setat utilizând *setJMSDeliveryMode(int deliveryMode)* la momentul când mesajul este produs *(TopicPublisher sau QueueSender).* Odată ce modul de livrare este setat la nivel clasei *MessageProducer,* el este aplicat tuturor mesajelor livrate de respectivul producător. Valoarea implicită este PERSISTENT.

```
// setarea modului de livrare JMS la nivelul producătorului
TopicPublisher topicPublisher = TopicSession.createPublisher(topic);
topicPublisher.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

JMSMessageID - Acest element de antet este un ***String*** a cărui valoare identifică în mod unic un mesaj în cadrul sistemului de mesagerie. *JMSMessageID* poate fi foarte util atunci când un consumator doreşte să arhiveze toate mesajele pe care le-a primit şi prelucrat, iar aceste mesaje trebuie să fie indexate de o manieră unică. Utilizat în conexiune cu *JMSCorrelationID, JMSMessageID* este de asemenea util pentru a pune în corespondenţă mesajele de cerere cu cele de răspuns, în cazul unei comunicări asincrone de tip cerere/răspuns.

```
String messageID = message.getJMSMessageID();
```

JMSTimestamp - Acesta este în mod automat setat de către producătorul mesajului în momentul când invocă metoda *send(). JMSTimestamp* conţine timpul la care mesajul a fost recepţionat de serverul de mesagerie şi nu timpul la care a fost de fapt livrat de producător. Acest element de antet este util pentru determinarea duratei timp scurse de la momentul la care mesajul a fost trimis către server şi momentul la care mesajul a fost consumat. Este reprezentat printr-o valoare de tip *long,* care păstrează timpul în milisecunde (scurse de la 1 ianuarie 1970).

```
long timestamp = message.getJMSTimestamp();
```

JMSExpiration - Prin setarea momentului la care un mesaj este considerat expirat se poate previne livrarea către consumatori a unui mesaj care nu mai este de actualitate, din perspectiva informaţiei ce o poartă. Este util pentru mesaje ale căror componentă de date de business este validă pentru o perioadă limitată de timp.

```
long timeToLive = message.getJMSExpiration();
```

Timpul până la expirarea mesajului este setat în milisecunde de către producător (de exemplu *TopicPublisher)* utilizând metoda *setTimeToLive().*

```
TopicPublisher topicPublisher = TopicSession.createPublisher(topic);
// setare timp de viaţă la 1 oră (1000 milisecunde x 60 secunde x 60 minute)
topicPublisher.setTimeToLive(3600000);
```

Serverul de mesagerie va adăuga valoarea *timeToLive* la timpul sistemului şi va seta *JMSExpiration.* Implicit, *timeToLive* este zero, ceea ce indică faptul că mesajul nu va expira. Prin apelul metodei *setTimeToLive()* cu argumentul zero (0) se asigură faptul că mesajul este creat fără a avea o dată de expirare. Orice apel programatic direct al metodei *setJMSExpiration()* va fi ignorat la momentul la care mesajul este trimis.

JMSRedelivered - Acest element de antet indică faptul că mesajul se doreşte a fi retrimis către consumator. Este modelat ca o valoare booleană, când *JMSRedelivered* este *true* atunci mesajus este retransmis, când este *false* nu se realizează acest lucru. Un mesaj poate fi marcat pentru retransmitere atunci când consumatorul a eşuat să confirme primirea acestuia la livrarea anterioră, sau când serverul de mesagerie nu este sigur dacă cel ce este de presupus să consume mesajul l-a primit deja sau nu.

```
boolean isRedelivered = message.getJMSRedelivered();
```

JMSPriority - Serverul de mesagerie poate asigna o anumită prioritate unui mesaj atunci când acesta este transmis în sistem. Există două categorii de priorităţi asociate mesajelor:

- nivelurile de la 0 la 4 modulează gradual o prioritate normală;
- nivelurile de la 5 la 9 modulează gradual o prioritate de rangul urgenţă/alertă (*expedited*).

Serverul de mesagerie poate utiliza prioritatea mesajului pentru a da întâietate unor mesaje care trebuie să ajungă mai rapid la consumatori. Mesajele din categoria urgenţă/alertă au întâietate faţă de cele cu o prioritate normală.

```
int priority = message.getJMSPriority();
```

Nivelul de prioritate asociat unui mesaj poate fi declarat de către un client JMS prin utilizarea metodei **setPriority()** la nivel producătorului mesajului.

```
TopicPublisher topicPublisher = TopicSession.createPublisher(topic);
topicPublisher.setPriority(9);
```

Orice apel programatic direct al metodei **setJMSPriority()** va fi ignorat la momentul transmiterii mesajului.

Elementele antetului asignabile de către programator

Chiar dacă cele mai multe elemente ale antetului unui mesaj JMS sunt asignate automat de către serverul de mesagerie la momentul transmiterii mesajului, unele pot fi totuşi setate explicit la nivelul obiectului **Message** înainte ca acesta să fie livrat de către producător.

JMSReplyTo - În unele cazuri, producătorul unui mesaj JMS poate dori ca aplicaţia care va consuma mesajul să trimită înapoi un mesaj de răspuns (de exemplu atunci când se foloseşte un mecanism de cerere/răspuns prin intermediul platformei de mesagerie). Elementul de antet *JMSReplyTo*, care conţine un obiect de tip *javax.jms.Destination*, indică care este adresa la care consumatorul JMS va trebui să trimită răspunsul. Utilizarea acestei proprităţi de la nivelul antetului, oferă posibilitate unei decuplări efective a producătorului mesajului de consumatorul acestuia, în cadrul scenariului de cerere/răspuns.

JMSType - Este un element opţional al antetului, care este setat de clientul JMS. Menirea lui este aceea de a identifica structura mesajului şi tipul de date de business. Trebuie remarcat faptul că acest element al antetului nu indică ce clasă de mesaj este trimisă (*MapMessage, TextMessage, ObjectMessage* etc.) ci, mai curând, un tip de înregistrare ce este folosit de componenta de persistare a serverului de mesagerie.

Atributele unui mesaj JMS acţionează ca elemente de caracterizare suplimentară a mesajului, pe lânga cele conţinute de antet. Ele permit programatorului să adauge mai multe informaţii structurate cu privire la conţinutul mesajului. Sunt, de asemenea, utilizate pentru a expune date de identificare ce pot fi ulterior prelucrate de componentele de selectare a mesajelor, atunci când se doreşte filtrarea acestora. Interfaţa *Message* oferă mai multe metode de acces şi modificare a acestor atribute. Valoare luată de un atribut poate fi de următoarele tipuri: *String, boolean, byte, short, int, long, float* sau *double.*Există trei categorii de atribute care se pot asocia unui mesaj JMS:atribute specifice aplicaţiei;atribute definite de către interfaţa JMS; atribute specifice serverului de mesagerie.

Java Mesage Service defineşte şase tipuri de interfeţe Message care trebuie să fie oferite de furnizorii de mesagerie compatibilă JMS. Implementare acestor interfeţe este însă lăsată la latitudinea producătorilor. Cele şase interfeţe sunt formate din interfaţa Message şi cinci subinterfeţe ale sale: TextMessage,StreamMessage,MapMessage,ObjectMessage,BytesMessage

Message - Tipul cel mai simplu de mesaj JMS este modelat de javax.jms.Message şi serveşte şi ca interfaţă de bază pentru celelalte interfeţe. Tipul Message poate fi creat şi folosit ca mesaj JMS şi fără a purta conţinut de business (*payload*). Acest tip de mesaj conţine numai antetul şi atributele (proprietăţile) JMS, fiind utilizat pentru notificarea de evenimente. Notificarea unui eveniment care a avut loc în sistem poate îmbrăca forma unui *broadcast*, a unei avertizări, sau a unei note informative cu privire la evenimentul care a avut loc

JMS nu este un simplu serviciu de coordonat evenimente. Prin mecanismele de procesare asincronă, de persistare şi livrare ulterioară garantată, furnizează funcţionalităţi care oferă aplicaţiilor pe care le deserveşte posibilitatea de a rula într-o manieră continuă, fără întreruperi pentru mentenanţă. Oferă flexibilitate integrării de noi componente prin furnizarea unor modele de comunicare de tip *pub/sub* şi *p2p* într-o manieră asincronă. Datorită tranparenţei oferite în ceea ce priveşte localizarea componentelor şi a mecanismelor de administrare şi control, furnizează premisele unei arhitecuri distribuite robuste.

9.2.6. JGroups pentru mesagerie de grup

JGroups este un instrument software utilizat pentru comunicarea între diferite entități software (servere, procese) ale unui cluster de mașini, fiind un toolkit Java dezvoltat în contextul proiectului JBoss (www.jboss.org) , ce oferă suport pentru o comunicație în rețea de tip multicast. Comunicația multicast este la nivel de grup, ceea ce înseamnă ca un singur dispozitiv din rețea poate transmite simultan mesaje spre mai multe dispozitive din rețea. Jgroups realizează implementarea unor mecanisme de comunicație de grup la nivel aplicație, oferind suport de comunicație sigur și simplu direct dezvoltatorilor de aplicații.

Principalele caracteristici ale acestui sistem sunt: simplitate în crearea de noi clustere, în subscrierea sau părăsirea unui cluster, detectarea membrilor acestuia și generarea de notificări referitor la acțiunile nodurilor componente (subscriere, părăsire, esec), detecția și îndepărtarea nodurilor care au întâmpinat o cădere. Însă, cel mai mare avantaj al sistemului JGroups, ce conduce la comunicare rapidă între nodurile unui cluster este *flexibilitatea stivei de protocoale*. Această caracteristică permite utilizatorilor să își configureze propria stivă de protocoale necesară unei aplicații sau chiar să își definească propriile protocoale de comunicație în funcție de necesitățile aplicației pe care o dezvoltă.

Caracteristici ale stivei de protocoale :

- Protocoale de transport: UDP (IP multicast) sau TCP
- Fragmentarea mesaje mari
- Retransmiterea mesaje

- Detectarea căderilor nodurilor
- Controlul debitului de mesaje pentru a preveni receptoare lente sa fie blocate de expeditori rapizi
- Protocoale de ordonare a mesajelor integrate: FIFO, Total Order
- Criptare/comprimare mesaje

Caracteristici funcționale toolkit :

- Crearea și ștergerea unui grup. Noduri din cluster pot fi răspândite în retea LAN sau WAN
- Intrarea în si parasirea unui grup (cluster de noduri).
- Detectare membrilor unui grup și notificarea lor cu privire la aderarea / parasirea / prăbușirea unuia dintre nodurile clusterului
- Detectarea și eliminarea nodurilor căzute
- Trimiterea și primirea de mesaje nod-la-grup (punct-la-multipunct) și trimiterea și primirea de mesaje nod-la-nod(punct-la-punct)

a)Modelul de sistem

JGroups este sistem software destinat comunicării între diverse noduri sau grupuri de noduri. Prin nod, se întelege un proces care se execută pe o anumită mașină fizică, astfel comunicarea se poate realiza între procese de pe mașini diferite, în manieră simplă. Fiecare proces poate să aparțină unui grup de comunicare. Un astfel de grup de comunicare este numit în continuare cluster și este identificabil prin numele propriu ,definit de către utilizator. Un grup nu trebuie creat în mod explicit, deoarece în momentul în care un proces dorește să se înscrie într-un anumit grup, iar acesta nu există, *grupul este creat de către sistem*. Fiecare proces poate solicita apartenența la orice grup, poate tirmite/primi mesaje tuturor membrilor grupului sau numai anumitor membri. Sistemul JGroups menține evidența tuturor membrilor afiliați la un anumit grup și notifică fiecare membru asupra modificărilor care au loc în structura grupului. Astfel că, în momentul în care un membru al grupului, părăsește/atașează grupului sau întâlnește o situație de eșec, ceilalți membri ai grupului sunt notificați asupra evenimentului respectiv.

Arhitectura sistemului JGroups este formată din trei componente principale care susțin transmiterea de mesaje. Prima componentă este *canalul de comunicare*, denumit JChannel, care este principala modalitate folosită de către dezvoltatori pentru a realiza comunicarea între procese. Această componentă este conectată la *stiva de protocoale* , reprezentând cea de a doua componentă, conectată direct la rețeaua de transmisie a datelor. A treia componentă se numeste *Building Blocks* și reprezintă un nivel de abstractizare mai înalt decât canalele, util pentru aplicații mai complexe.

Stiva de protocoale conține un set de protocoale care participă atât la procesul de primire/recepționare a mesajelor cât și la cel de transmitere. In momentul în care o aplicație client doreste să transmită un mesaj, acesta este transmis printr-un canal, acesta fiind conectat la o stivă de protocol. Ori de câte ori aplicația trimite un mesaj, canalul il transmite spre stiva de protocoale, care il transmite către protocolul din vârful stivei. Protocolul procesează mesajul și il transmite corespondent în stivă către protocolul de transport și acesta il pune în rețea. Același lucru se întâmplă în sens invers la receptor : protocolul de transport ascultă pentru mesajele de pe rețea, atunci când un mesaj este primit acesta va fi transferat stivei de protocoale până când ajunge la canal . Atunci când o aplicație se conectează la canal, stiva de protocol va fi pornită, iar ala deconectare va fi oprită apoi distrusă eliberând resursele.

Fiecare proces care doreste să se alăture unui grup de comunicare și să transmită mesaje către membrii grupului trebuie să creeze un astfel de canal de comunicare. Dacă mai multe procese creează canale de comunicare cu același nume, acestea vor constitui un grup, astfel, canalul este modul prin care un proces comunica cu celelalte procese. Dacă procesul nu mai dorește sa fie afiliat la un anumit grup, acesta poate semnala deconectarea de la canalul creat, ceea ce este echivalent cu părăsirea grupului de comunicare. Ficare canal menține o listă cu procesele care sunt conectate la acelasi grup, care se numește *View*. De fiecare dată când un nou proces se alătură grupului de comunicare sau părăsește grupul de comunicare, se va crea un nou view care conține modificările care au avut loc. Un proces poate selecta un alt proces din View căruia să îi transmită mesaje în mod unicast, sau poate să transmita mesaje tuturor proceselor în mod multicast.Canalele pot fi configurate prin fișiere XML, dar este permisă configurare și prin arbori DOM, URI-uri sau programatic.

b)API JGroups

Principalul API este reprezentat de clasa org.jgroups.JChannel a cărei interfață este prezentată mai jos :

```
Public class JChannel extends Channel{
Public JChannel (String properties)throws ChannelException;
Public void setreceiver(receiver r);
Public void connect (String cluster_name) throws channelException;
Public void send(Message msg)throws ChannelException;
Public View get View();
Public address getlocaladdress();
Public void disconnect();
Public void  close();}
```

Pentru crearea/conectarea la un cluster se folosește metoda connect().Toate canalele având aceeași configurație si același nume de cluster vor face parte din același cluster. In continuare sunt descrise funcțiile clasei Channel:

- *send()* – trimite un mesaj în cadrul cluster-ului. Dacă adresa de destinație este null acesta va fi trimis tuturor membrilor cluster-ului, în caz contrar va fi trimis membrului cu adresa specificată.
- *disconnect()* – invocata de un nod pentru a părăsi cluster-ul
- *close()* – distruge canalul. Un canal închis nu mai poate fi redeschis. In cazul în care canalul nu este deconectat de la cluster, realizează și deconectarea acestuia
- *getLocalAddress()* – returnează adresa locala a nodului

Un view este o reprezentare a nodurilor prezente la un moment dat în cluster. In cadrul unui view nodurile sunt ordonate în funcție de momentele de timp la care au intrat în cluster (cel mai vechi este întotdeauna primul). Pentru a obține view-ul curent se folosește metoda *getView()* – returnează view-ul curent

Pentru efectuarea unor acțiuni, în momentul recepționării mesajelor se atribuie canalului un Receiver folosind funcția setReceiver(). Acesta are următoarea interfață:

```
Void receive(Message msg);
Void viewAccepted(View new_view);
```

Cele două funcții pot fi utilizate după cum urmează :

- *receive()* – este apelata in momentul recepționării unui mesaj
- *viewAccepted()* – este apelata când un nod se alătură sau părăsește cluster-ul

Structura unui mesaj este prezentată în continuare.Acesta conține adresa sursa, adresa destinație precum și un byte[] care reprezintă mesajul .

```
Public class Message implements Streamable{
Protected Address dest_addr=null;
Protected Address src_addr=null;
Private byte [] buf=null;
Public byte[] getBuffer();
Public void setBuffer(byte[] b);}
```

9.3.Exemple

9.3.1.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului JMS exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/09_JMS

9.3.2. Aplicaţie de chat bazat pe modelul pub-sub din JMS

Aplicaţia este destinată comunicării de mesaje între mai mulţi clienţi în sistem tip chat, folosind modelul *publish/ subscribe*. Aceasta foloseşte pachetul Java JMS pentru transmisia şi recepţia mesajelor, pasarea acestora fiind intermediată de broker-ul JMS care trebuie să ruleze ca serviciu.

Aplicaţia este compusă din următoarele module:

AplicatieChat – conţine funcţia main() şi codul de iniţializare al ferestrei,

PanouChat – panou care conţine controalele folosite la afişarea mesajelor, *dialogInfoConexiune* – căsuţa de dialog care permite selectarea parametrilor de conectare,

interfaţa *CreatorMesaje*, şi clasele ce implementează această interfaţă – *CreatorMesajText* şi *CreatorMesajStream*, folosite la crearea mesajelor de tip text şi stream, încapsulând informaţii care identifică unic mesajul cum ar fi expeditorul şi tipul acestuia.

Pentru trimiterea mesajelor, aplicaţia foloseşte clase derivate din clasa Message a pachetului JMS – *TextMessaje* şi *StreamMessage*.

Tipurile de mesaje suportate sunt:

INTRARE – intrare în sesiunea de chat,

NORMAL – mesaj propriu-zis,

IESIRE – ieşire din sesiunea de chat.

Fiecare mesaj conţine, pe lânga textul propriu-zis şi câmpuri care identifică expeditorul şi tipul mesajului. Codificarea acestor câmpuri se face fie folosind proprietăţi ale clasei *Message* (*CreatorMesajText*), fie codificând aceste câmpuri în cadrul unui stream (*CreatorMesajStream*).

Când aplicaţia intră într-o sesiune, transmite un mesaj broadcast INTRARE. Toţi clienţii aflaţi în sesiune vor primi acest mesaj şi fereastra va afişa intrarea noului client. Odată ce aplicaţia a intrat în sesiune, mesajele trimise sunt codificate ca obiecte *Message*. La transmiterea acestor mesaje (publicare), toţi clienţii vor afişa în fereastra de mesaje numele expeditorului şi mesajul. Când un client se deconectează, se transmite un mesaj IESIRE.. Pentru încapsularea numelui expeditorului şi a tipului mesajului, s-au folosit două metode de codificare: clasa *CreatorMesajText* foloseşte proprietăţile puse la dispoziţie de clasa JMS *TextMessage*, asociind valoarea corespunzătoare unui nume de proprietate, clasa *CreatorMesajStream* scrie cele două informaţii direct în streamul de ieşire, după acestea fiind scris mesajul efectiv.

La evenimentul de primire a unui mesaj, citirea trebuie făcută în aceeaşi ordine, în caz contrar datele fiind inconsistente. Clasa JMS folosită pentru codificarea acestui tip de mesaj este *StreamMessage*. Pentru trimiterea mesajului, acesta este creat printr-o metoda de tip *factory* folosind una din cele două clase mai sus menţionate, mesajul rezultat fiind trimis efectiv prin publicarea acestuia folosind un obiect JMS de tip *TopicPublisher*, prin metoda *publish()*.

Pentru ca şi clienţii să primească mesajele trimise în sesiunea de chat, aceştia trebuie să-şi creeze un obiect de tip *TopicSubscriber*, pentru care trebuie să stabilească o clasă ce implementează interfaţa JMS *MessageListener*. Aceasta clasă ascultă mesajele primite din sesiune definind metoda *onMessage()* în care aplicaţia preia mesajul şi îl afişează în interfaţă.

Succesiunea de operaţii necesare pentru intrarea într-o sesiune de chat este:

1. crearea unui obiect de tip conexiune apelând metoda factory *createTopicConnection()* a clasei *TopicConnectionFactory*,
2. crearea unui obiect de tip sesiune apelând metoda factory *createTopicSession()* a clasei *TopicConnection*,
3. crearea unui publisher apelând metoda de tip factory *createPublisher()* a clasei *TopicSession*,
4. crearea unui subscriber apelând metoda de tip factory *createSubscriber()* a clasei *TopicSession*,
5. stabilirea unei clase care ascultă mesajele din sesiune apelând metoda *setMessageListener()*,
6. activarea conexiunii apelând metoda *start()* a clasei *TopicConnection*.

După această succesiune de paşi, clientul îşi anunţă prezenţa în sesiune trimiţând un mesaj broadcast INTRARE.

Folosirea tehnologiei JMS a permis crearea unei aplicaţii în care clienţii comunică prin mesaje, publicându-le într-un sistem unificat. Între clienţi cuplarea este slabă, aceştia operând doar cu un obiect de tip sesiune care se ocupă de gestionarea trimiterii mesajelor în mod asincron, nefiind necesare mecanisme de confirmare a recepţiei. Broker-ul JMS se ocupă de funcţionarea sesiunilor existente în aplicaţiile care utilizează această tehnologie, existând o conexiune permanentă cu acesta.

Mesajele trimise se pot codifica în diverse formate, cu ajutorul claselor puse la dispoziţie de pachetul JMS în acest scop. Dintre acestea amintim: text simplu, stream, şir de octeţi, mesaje care încapsulează obiecte etc. Aplicaţia funcţionează în sistem *publish/ subscribe*, în care fiecare client publică în sesiune mesaje de diferite tipuri, acestea fiind recepţionate de ceilalţi clienţi conectaţi şi interpretate în funcţie de tip (mesaje de intrare în sesiune, de ieşire din sesiune şi mesaje propriu-zise). Obiectul JMS care identifică un grup de discuţii este definit de clasa *Topic*, cu ajutorul căruia mesajele trimise sunt direcţionate doar către clienţii care sunt conectaţi la acelaşi grup, făcându-se efectiv o distribuire a acestora în funcţie de topic.

9.3.2. Aplicație SimpleChat implementată cu tehnologia JGroups

Simple Chat este o aplicatie care realizează transmiterea unor mesaje între membrii unui grup de comunicare cât și managementul acestuia. Toate instanțele aplicației Simplechat se vor regăsi între ele și vor forma un cluster. Nu este necesară rularea unui server de chat la care celelalte instanțe să se conecteze. Prin urmare nu este nici un punct de eșec în cadrul sistemului. Un mesaj din chat este trimis spre toate instanțele clusterului. O instanță primește o notificare când o alta instanță părăsește sau se alătura grupului. La execuție se creeaza un nou obiect JChannel prin care se transmit mesaje către toți membrii subscriși unui cluster. O instanță Jchannel este creată cu o anumită configurație care definește proprietățile canalului. In clasa SimpleChat se declară un canal în cadrul metodei start. Prin intermediul acestui canal se va realiza transmisia de mesaje. Prin intermediul metodei connect se va realiza și conectarea la clusterul cu numele dat ca și argument. In cazul în care clusterul nu există ,acesta va fi creat de către sistem.

```
private void start() throws Exception {
      channel=new JChannel();
      channel.setReceiver(this);
      channel.connect("ChatCluster");
      channel.getState(null, 10000);
      eventLoop();
      channel.close();}
```

Se crează canalul folosind constructorul empty. Acest constructor creează canalul și îi atribuie proprietățile default. Se poate crea un canal cu anumite proprietăți prin parsarea unui fisier xml ca argument la constructorul Jchannel(de exemplu new JChannel("/home/xxx/udp.xml").

Implementarea buclei principale a clusterului este efectuata prin intermediul funcției :

```
private void eventLoop() {
        BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            try {
                System.out.print("> "); System.out.flush();
                String line=in.readLine().toLowerCase()ț
                if(line.startsWith("quit") || line.startsWith("exit")) {
                    break;}
                line="[" + user_name + "] " + line;
                Message msg=new Message(null, null, line);
                channel.send(msg); }
            catch(Exception e) {} } }
```

In cadrul acestei metode se citește un mesaj de la tastatură. Dacă mesajul nu este quit sau exit, va fi trimis prin intermediul canalului spre toate nodurile din cluster. Apelarea acestei bucle blochează canalul până când un mesaj este introdus. Primul argument al constructorului Message este adresa destinatie. O adresă destinţtie nulă înseamnă că mesajul va fi trimis spre toate nodurile din cluster. Daca adresa destnație nu e nulă, mesajul este trimis doar spre acea adresț.Al doilea argument este adresa la care se trimite mesajul. Stiva de protocol va insera in mod automat adresa. Al treilea argumentul este linia de pe care se citește mesajul.Cu ajutorul serializării JAVA se crează un buffer de tip byte care conține payload-ul mesajului.

Pentru ca sa poata receptiona un mesaj, clasa trebuie sa extinda ReceiverAdapter. Receiver-ul trebuie setat in metoda start(): channel.setReceiver(this);

ViewAccepted este apelată de fiecare dată când un nou nod este adaugat clusterului sau cand o instanta paraseste clusterul. Metoda toString afiseaza id-ul vederii si o lista a instantelor curente din cluster. In metoda receive() se primeste ca si argument un mesaj. Din acest mesaj se ia buffer-ul ca si un obiect si se afiseaza in consolo continutul lui. Se printează de asemenea adresa de la care a fost trimis mesajul.

```
public void viewAccepted(View new_view) {
        System.out.println("** view: " + new_view); }

    public void receive(Message msg) {
        String line=msg.getSrc() + ": " + msg.getObject();
        System.out.println(line);
        synchronized(state) {state.add(line);} }
```

Unul din cazurile de utilizare a JGroups este menținerea stării replicată intr-un cluster. De exempluo stare ar putea fi reprezentata de toate sesiunile HTTP dintr-un server de web. Dacă aceste sesiuni sunt reproduse pe un cluster, atunci clienții pot accesa orice server din cluster, după ce un server care a găzduit sesiunea clientului s-a prăbușit, iar sesiunile utilizator vor fi în continuare disponibile.

Transferul de stare în JGroups se face prin implementarea a două metode (getState () și setState ())și apelarea metodei JChannel.getState (). Rețineți că, pentru a putea folosi transferul de stare într-o aplicație, stiva de protocol trebuie să aibă un protocol de transfer de stare. Primul argument al metodei getState este instanța țintă, iar NULL înseamnă ca va lua starea de la prima instanță (coordonatorul). Al doilea argument este timeout; aici suntem dispuși să aștepte timp de 10 secunde pentru a transfera o stare. În cazul în care starea nu poate fi transferată în acest moment, atunci o excepție va fi aruncată. Valoarea 0 pentru acest argument înseamnă o așteptare fără limita de timp.

ReceiverAdapter definește metoda getState care este apelata pe o instanță existentă pentru a reda starea clusterului.

```
public void getState(OutputStream output) throws Exception {
        synchronized(state) {
            Util.objectToStream(state, new DataOutputStream(output)); }
```

setState are drept scop acela de a citi starea fluxului de intrare :

```
@SuppressWarnings("unchecked")
    public void setState(InputStream input) throws Exception {
        List<String>            list=(List<String>)Util.objectFromStream(new
DataInputStream(input));
        synchronized(state) {
            state.clear();
            state.addAll(list);}
        System.out.println("received state (" + list.size() + " messages in
chat history):");
        for(String str: list) {System.out.println(str); }
```

9.4. Întrebări teoretice

9.4.1.Prin ce diferă cele două modele conceptuale de mesagerie JMS Ș p2p și pub-sub ?
9.4.2.Ce reprezintă o sesiune ? dar un topic ?
9.4.3.Care este cel mai important avantaj oferit de tehnologia JGroups ?
9.4.4. Identificați proprietățile ce pot fi asignate de programator pentru mesajele JMS.

9.5. Probleme propuse

9.5.1.Implementați câte un Sistem de chat folosind cele două tehnologii JMS și respectiv JGroups, realizând soluții echivalente funcțional care să poată fi ulterior comparate din perspectiva complexității codului.

9.5.2.Realizați utilizând facilitățile oferite de toolkitul JGroups o aplicație minimală de tip Whiteboard.

9.5.3.Propuneți o implementare JGroups pentru un Sistemul de distribuție a taskurilor , capabil să permită managementul acestora la nivelul unui *cluster logic* ,implementat utilizând virtualizarea oferită de Docker.

9.5.4.Să se implementeze simularea unui *Sistem distribuit de cacheuri* , imaginând aplicarea sa pentru un cluster de noduri.Un schelet de rezolvare a cerinței poate fi consultat la adresa http://www.jgroups.org/replcache.html

9.6. Referințe bibliografice

1. Specificația JMS
   https://www.oracle.com/technetwork/java/jms/index.html
2. Tutorial JMS :
   https://howtodoinjava.com/jms/jms-java-message-service-tutorial/
3. Exemplu de implementare mecanism Producer-Consumer bazat pe JMS
   https://docs.oracle.com/javaee/6/tutorial/doc/bncfa.html
4. JMS în Spring https://spring.io/guides/gs/messaging-jms/
5. Tutorial JGroups : http://www.jgroups.org/tutorial/pdf/tutorial.pdf
6. Manual JGroups : http://www.jgroups.org/manual/pdf/manual.pdf

10. Servicii de mesagerie distribuită

Orice sistem de mesagerie respectă o arhitectură specifică, deseori fiind vorba de modelul *publish-subscribe*. Aplicațiile care comunică printr-o paradigmă de tip publish & subscribe necesită existența unor expeditori care să trimită mesaje fără a preciza în mod explicit destinatarii (fără a avea cunoștințe referitor la aceștia), repsetiv existența unor destinatari sau abonați (subscribers) care să primească acele mesaje trimise de expeditori (publish-eri), pentru care au manifestat înteres înregistrându-se/abonându-se la aceșia. Această decuplare între expeditori și destinatari este, de obicei, realizată de o entitate care se interpune între aceștia și care servește ca nivel intermediar (engl. middleware) , reprezentând un serviciu cu rol de gestiune și eventual stocare a mesajelor.

10.1.Obiective

- Studiul modelului *publish-subscribe* integrat protocolului AMQP
- Studiul unui instrument software care facilitează construirea unei arhitecturi MOM –RabbitMQ pentru a oferi servicii de mesagerie distribuită
- Implementarea unor modele de aplicații distribuite care să realizeze comunicare asincronă între mai mulți clienți

10.2.Concepte

Message Oriented based Middleware (MOM) este o infrastructură software care suportă trimiterea și primirea de mesaje între sistemele distribuite, respectând, în acest sens, paradigma arhitecturală *publish-subscribe*. Un aspect extrem de important al soluțiilor de tip MOM este acela că permit serviciilor software să fie distribuite pe platforme eterogene și în acest fel reduc complexitatea dezvoltării unor aplicații care ar necesita integrarea complexă a mai multor sisteme de operare și protocoale de rețea. Astfel, MOM reprezintă un strat de comunicare distribuit care izolează dezvoltatorul software de detaliile diferitelor sisteme de operare și interfețe de rețea, furnizându-i API-uri extensibile pe diferite platforme și rețele pentru programarea aplicațiilor distribuite ce pot fi conceptualizate ca sisteme de mesaje.

10.2.1. Protocolul și arhitectura AMQP

Advanced Message Queuing Protocol (AMQP), este un protocol de mesagerie care permite aplicațiilor/platformelor să comunice cu un middleware construit pe baza unor *brokeri de mesagerie*.

Acești brokeri de mesagerie primesc mesaje de la expeditori (publishers), reprezentanții acestora fiind cunoscuți drept producători sau aplicații care trimit mesaje. Brokerii vor direcționa mesajele către destinatari sau consumatori (subscriber**s**), reprezentați prin aplicații care vor procesa informațiile primite. De menționat este faptul că AMQP fiind un protocol de rețea, implică ca producătorii, consumatorii și brokerii să fie capabili să ruleze ca enitități diferite pe mașini fizice diferite.

Modelul AMQP funcționează astfel:

- Producătorii vor trimite mesaje către brokeri. În cadrul broker-ului, acestea vor fi preluate de către exchangeri, care sunt adesea comparați cu oficiile poștale sau cutiile poștale. Exchangerii distribuie, apoi, copii de mesaje unor cozi de mesaje, folosind reguli numite *bindings.*
- Brokerul livrează mesajele către consumatorii abonați la cozile de așteptare sau consumatorii preiau mesaje de la cozile de mesaje.

La publicarea unui mesaj, publisherii pot specifica diferite atribute ale mesajelor (metadate pentru mesaje). Din toate informațiile transmise broker-ului, meta-datele despre mesaje pot fi folosite

de către acesta, iar conținutul explicit al mesajului este complet invizibil brokerului și este folosit numai de aplicațiile care primesc mesajul.

Sistemele distribuite se confruntă cu o provocare continuă în ceea ce privește fiabililtatea. Astfel, în contextul MOM, aplicațiile ar putea să nu reușească să proceseze mesajele datorită unor erori de transmisie ori de recepție (deseori recunoscute ca timed-out), prin urmare modelul AMQP integrează un mecanism de tip achitare a mesajelor (*message acknowledgements)*: atunci când un mesaj este transmis unui consumator, consumatorul notifică brokerul printr-o confirmare de primire a mesajului. De menționat faptul că un broker va elimina complet un mesaj dintr-o coadă doar atunci când primește o notificare pentru acel mesaj ,sau grup de mesaje.

Tipuri de interacțiuni

Exchangerii sunt entități în care sunt trimise mesajele, iau un mesaj și îl direcționează către zero sau mai multe cozi. Algoritmul de rutare utilizat depinde de tipul de exchanger și de regulile numite *binding-uri.* Brokerii AMQP 0.10.1 oferă patru tipuri de exchangeri:

NUME	NUME IMPLICIT - PREDECLARAT
Direct exchanger	(String empty) și amq.direct
Fanout exchanger	amq.fanout
Topic exchanger	amq.topic
Headers exchanger	amq.match (și amq.headers în RabbitMQ)

Tabel 10.1. Tipuri de exchangeri

Default Exchanger .Exchanger-ul default reprezintă exchangerul implicit, fără nume, pre-declarat de broker. Acesta are o proprietate specială care îl face foart util pentru aplicații simple: fiecare coadă creată este legată automat de exchanger-ul default printr-o cheie de rutare care este identică cu numele cozii.

Direct Exchanger .Un Direct Exchanger oferă mesaje către cozi folosind o cheie de rutare specifică. Un exchanger direct este ideal pentru *rutarea unicast* a mesajelor (deși acestea pot fi folosite și pentru *rutarea multicast*). Iată cum funcționează:

- Coada se leagă de exchanger cu o cheie de rutare K;
- Atunci când un mesaj nou cu cheia de rutare R ajunge la exchanger-ul direct, acesta îl direcționează către coadă doar dacă K = R.

Fanout Exchanger . Acest tip de exchanger direcționează mesaje către toate cozile care sunt legate de el, ignorând cheile de rutare. Dacă avem N cozi legate de un Fanout Exchanger, atunci când un nou mesaj este publicat, adică trimis către acest exchanger, o copie a mesajului este transmisă tuturor celor N cozi legate de el.

Topic Exchanger .Un Topic Exchanger direcționează mesajele către una sau mai multe cozi nu doar în funcție de cheia de rutare, ci și în funcție de pattern-ul care a fost folosit pentru a lega coada de exchanger. Topic Exchanger-ii sunt frecvent utilizați pentru *rutarea multicast* a mesajelor. Topic Exchanger-ii au un set foarte larg de cazuri de utilizare. Ori de câte ori o problemă implică mai mulți consumatori/aplicații care aleg în mod selectiv ce tip de mesaje doresc să primească, trebuie luată în considerare utilizarea Topic Exchanger-urilor.

Headers Exchanger .Acesta este proiectat pentru rutarea ce utilizează mai multe atribute care sunt mai ușor de exprimat ca header de mesaje decât ca o cheie de rutare. Headers Exchangerii ignoră atributul reprezentativ al cheii de rutare. În schimb, atributele utilizate pentru rutare sunt preluate din atributul headers. Un mesaj este direcționat spre o coadă dacă valoarea header-ului este egală cu valoarea specificată la legare cu acea coadă.

Cozile din modelul AMQP sunt foarte asemănătoare cu cozile din alte sisteme de mesagerie. Acestea stochează mesaje care sunt consumate, apoi, de aplicații. Cozile împărtășesc unele proprietăți cu exchangerii, dar au și unele proprietăți suplimentare. Înainte de a putea fi utilizată o coadă, aceasta trebuie declarată. Declararea unei coade va determina crearea acesteia dacă aceasta nu există deja. Declararea unei cozi care există deja nu va avea efect, dacă atributele sale sunt identice cu cele din ale altei cozi.

Binding-urile sunt reguli pe care exchangerii le utilizează (printre altele) pentru a direcționa mesajele către cozile de mesaje. Pentru a instrui un exchanger E să direcționeze mesaje către o coadă Q, trebuie să existe un binding între Q și E. Binding-urile pot avea un atribut opțional care specifică cheia de rutare, utilizat de unele tipuri de exchanger-i. Scopul cheii de rutare este acela de a selecta anumite mesaje publicate/trimise către un exchanger și care urmează să fie direcționate către o coadă. Cu alte cuvinte, cheia de rutare acționează ca un filtru. Dacă mesajul transmis prin protocolul AMQP nu poate fi direcționat către nicio coadă (de exemplu, pentru că nu există vreun binding al exchanger-ul spre care a fost trimis), acesta este fie abandonat și pierdut, fie returnat celui ce a inițiat transmisia mesajului prin AMQP, în funcție de atributele mesajului pe care acesta le-a setat.

Stocarea mesajelor în cozi este inutilă, cu excepția cazului în care aplicațiile le pot consuma, adică să există entități (consumatori) care vor utiliza mesajele din coadă. În modelul AMQP 0-9-1, există două metode pentru ca aplicațiile să facă acest lucru:

- Să fie trimise mesaje către consumatori („push API”);
- Consumatorii să ceară mesaje atunci când există această nevoie („pull API”).

În cazul "push API", aplicațiile trebuie să indice interesul de a consuma mesaje dintr-o anumită coadă. Atunci când fac acest lucru, spunem că aceștia înregistrează un consumator la o coadă sau, pur și simplu, se înscriu la o coadă. Fiecare consumator are un identificator, sub formă de șir de caractere, numit etichetă de consum. Aceasta poate fi folosită pentru a vă șterge înregistrarea efectuată pentru o anumită coadă.

Metode AMQP

AMQP este un protocol structurat, având un anumit număr de metode bine definite. Metodele sunt operații (cum ar fi metodele HTTP) și nu au nimic în comun cu metodele din limbajele de programare orientate pe obiecte. Metodele AMQP sunt grupate în clase. Clasele sunt doar grupări logice ale metodelor AMQP.

Astfel, dacă considerăm clasa exchanger, vom găsi un set de metode legate de operațiunile cu exchanger-ii. De exemplu, o parte dintre ele sunt:

- *exchange.declare*
- *exchange.declare-ok*
- *exchange.delete*
- *exchange.delete-ok*

Aceste operații formează perechi logice: *exchange.declare* cu *exchange.declare-ok* și *exchange.delete* cu *exchange.delete-ok,* reprezentând request-uri ,trimise de clienți și responses ,trimise de brokeri ca răspuns la request-urile menționate mai sus (detalii suplimentare în documentul referință AMQP 0-9-1, care conține detalii complete despre toate metodele AMQP.)

Conexiunile sunt, de obicei, de lungă durată. AMQP este un protocol de nivel aplicație, care utilizează TCP pentru o livrare sigură. Conexiunile folosesc conceptele autentificării și pot fi protejate utilizând TLS. Atunci când o aplicație nu mai are nevoie să fie conectată la server, ar trebui să se închidă conexiunea AMQP 0-9-1 în locul închiderii bruște a conexiuneii TCP subiacentă.

Unele aplicații au nevoie de mai multe conexiuni către broker. Cu toate acestea, nu este recomandată păstrarea deschisă a mai multor conexiuni TCP în același timp, deoarece va exista un consum al resurselor de sistem. Conexiunile AMQP 0-9-1 sunt multiplexate cu ajutorul canalelor care pot fi considerate drept "conexiuni light-weight care partajează o singură conexiune TCP".

Fiecare operație efectuată de un client are loc pe un canal. Comunicarea pe un anumit canal este complet separată de comunicarea pe un alt canal, prin urmare, fiecare metodă are, de asemenea, un

ID (numărul canalului), un număr întreg pe care atât brokerul, cât şi clienţii îl folosesc pentru a determina pentru care canal va fi procesată operaţia/metoda.

Un canal există doar în contextul unei conexiuni şi niciodată pe cont propriu. Când o conexiune este închisă, toate canalele de pe ea sunt închise. Pentru aplicaţiile care utilizează mai multe thread-uri/procese, se obişnuieşte deschiderea unui nou canal pentru fiecare thread/proces, evitându-se partajarea unor canale între thread-uri/procese.

Pentru a permite unui singur broker să găzduiască mai multe medii izolate (grupuri de utilizatori, exchanger-i, cozi etc.), AMQP oferă conceptul de host-uri virtuale (vhosts). Acestea sunt similare cu host-urile virtuale folosite de serverele Web populare şi oferă medii complet izolate în care trăiesc entităţile AMQP. Clienţii AMQP specifică ce vhosts doresc să utilizeze în timpul „negocierii" conexiunii.

10.2.2. Frameworkul RabbitMQ

RabbitMQ (www.rabitmq.com) este o implementară cu largă utilizare şi peformantă astandardului AMQP , ce conţine complementar o serie de extensii , astfel :

- **Stocare în memorie şi durabilitate opţională (limitată)**

RabbitMQ stochează toate mesajele pe care le manipulează în memorie, oferind în acelaşi timp etichete care pot fi plasate pe cozi sau mesaje pentru a le marca durabile. Cu toate acestea, durabilitatea realizată nu este comparabilă cu cea pe care o oferă Apache Kafka, deoarece, în cazul RabbitMQ, aceasta este temporară, mesajele fiind scrise pe disc numai dacă nu există niciun consumator care să poată procesa imediat mesajul. Mesajele (indiferent de valoarea etichetei de durabilitate) pot fi eliminate de pe disc dacă memoria broker-ului este epuizată. Odată ce mesajul a fost livrat (sau consumarea lui a fost confirmată), mesajul este înlăturat, chiar dacă a fost etichetat ca durabil. RabbitMQ nu oferă o modalitate de a impune o stocare permanentă precum Apache Kafka şi, ca atare, nu permite consumatorilor să refolosească mesaje vechi.

- **Livrare fiabilă**

Pentru mesajele trimise către consumatori se poate solicita un mesaj de confirmare .Acest lucru asigură că mesajele au fost consumate cu succes înainte ca acestea să fie eliminate din memoria broker-ului. Implicit nu este nevoie de confirmări, iar livrarea încheiată cu succes către unul dintre consumatorii înregistraţi este suficientă pentru ca broker-ul să înlăture mesajul. Totuşi, mesajele de confirmare ajută la reducerea numărului de mesaje neprocesate, oferind în acelaşi timp oportunitatea consumatorului să notifice broker-ul că un mesaj nu a putut fi procesat. Pentru unele cazuri de utilizare precum o bază de date care devine temporar indisponibilă şi, prin urmare, împiedică modificarea acesteia, în timp ce conexiunea cu RabbitMQ nu întâmpină niciun eşec, aceasta caracteristiciă se poate dovedi a fi folositoare.

- **Cozi şi exchangeri declaraţi dinamic**

Chiar dacă administratorul broker-ului le poate crea în avans, noi cozi şi exchangeri pot fi creaţi direct în codul producătorilor şi consumatorilor. Aceasta este o practică foarte bună pentru a se asigura că mesajele nu sunt trimise către cozi şi exchangeri inexistenţi.

- **Gestionarea permisiunilor şi suport SSL**

RabbitMQ le cere producătorilor, consumatorilor şi administratoilor să se autentifice înainte de a putea efectua orice operaţiune pe broker. Pot fi acordate diferite permisiuni diferiţilor utilizatori, permiţând astfel aplicarea unui control eficient al accesului. SSL este, de asemenea, acceptat şi poate fi utilizat pentru autentificare şi pentru criptarea datelor transmise prin canale între broker şi producători şi consumatori.

10.3.Exemple

10.3.1.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului JMS exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/10_MOM

10.3.2.Aplicația ilustrativă a fost creată pentru a oferi un exemplu al utilizării sistemului RabbitMQ, abordând conceptele de comunicare bazată pe paradigma mesageriei – cel de transmitere a unor mesaje între procese printr-un *mecanism asincron.* Aceasta va putea fi folosită de către două tipuri de utilizatori, cei care sunt înregistrați în cadrul platformei – utilizatorii – și ce care vor vizita platforma – clienții. Aplicația permite oricărui client să devină abonat al platformei atunci când acesta, prin intermediul intefeței utilizator, accesează opțiunea de abonare. Prin această abonare, clientul va primi notificări prin e-mail atunci când unul din utilizatorii platformei vor trimite vreo notificare prin accesarea opțiunii de trimitere mail. Dacă se consideră inutilă abonarea, clientul abonat poate să acceseze opțiunea de dezabonare, prin care acesta va fi eliminat din lista de notificare prin e-mail. De asemenea, orice client are posibilitatea de a accesa opțiunea de autentificare ca utilizator în cadrul platformei. Menționăm faptul că utilizatorii sunt prestabiliți sau pot fi creați doar de către alți utilizatori.

Cazurile de utilizare specifice aplicației sunt structurate pe cele două categorii de utilizatori și anume Clientul (Subscribe/Unsubscribe/Authenticate) respectiv Userul ce extinde Clientul (Send email, Create newAccount/Logout) .

Transmiterea de mesaje s-a realizat folosind CloudAMQP care pune la dispoziție RabbitMQ sub forma unui serviciu în cloud. Printre cele mai importante caracteristici oferite de CloudAMQP putem enumera:

- Cozi de mesaje în cloud - cloudAMQP instalează și gestionează singur cluster-ele RabbitMQ. RabbitMQ suportă diferite protocoale precum AMQP, MQTT, HTTPS STOMP și WebSockets.
- Scalare - folosind CloudAMQP se poate scala cluster-ul fără întreruperi. Același lucru este valabil și pentru upgrade-ul server-ului la o nouă versiune Erlang sau RabbitMQ.
- RabbitMQ Diagnostic Tool - acest utilitar vine ca ajutor pentru identificarea erorilor comune în cluster-ul RabbitMQ. Instrumentul va verifica configurația și va oferi sugestii despre lucrurile care trebuie analizate.
- Invitarea membrilor de echipă - folosind CloudAMQP vă puteți invita colegii pentru a gestiona același set de instanțe de pe conturi diferite.
- Plug-in-uri - din panoul de control al CloudAMQP se pot activa cu ușurință plug-in-urile RabbitMQ pentru un anumit server.
- Noduri adiționale în cluster - pentru instanțe dedicate din AWS și Azure puteți specifica numărul de noduri de care aveți nevoie. Se poate, de asemenea, crea cluster-ul întru VPC dedicat.

Pentru rularea aplicației se recomandă crearea unui cont propriu pe **CloudAMQP** și setarea configurațiilor proprii în cadrul aplicației. Acest lucru se poate realiza prin urmărind pașii de mai jos:

1. Crearea unui cont la link-ul https://www.cloudamqp.com/;
2. Se va apăsa butonul **Create New Instance** și se vor urma pașii pentru crearea unei noi instanțe în cloud:
   a. Se va insera un nume pentru instanță (de exemplu, NotifierApp) și se va apăsa butonul Select Region;
   b. Se va selecta un Data Center din lista prezentată și se va apăsa butonul Review;
   c. În noua pagină deschisă se vor prezenta detaliile instanței pe care sunteți pe cale să o creați (câmpurile specificate sunt Name, Provider, Region, Backend și Tags). Se va apăsa butonul Create Instance pentru a finaliza operațiunea de creare.
3. Deschideți instanța pe care tocmai ați creat-o și identificați câmpurile Host(s), User & Vhost și Password. Câmpurile respective vor trebui înlocuite în codul aplicației în modulul common din clasa QueueConnectionFactory, astfel:

```
factory.setUsername("<User & Vhost>"); factory.setPassword("<Password>");
factory.setVirtualHost("<User & Vhost>"); factory.setHost("<Host(s)>");
```

Rularea aplicației necesită:

- Crearea unei baze de date MySQL cu denumirea notifier-db;
- Configurarea unui server de Apache Tomcat pentru executa modulul notifier din cadrul proiectului;
- Pornirea consumatorului prin executarea metodei main din cadrul modului consumer și a clasei ConsumerStart.

Exemplele prezentate mai jos se bazează pe faptul că RabbitMQ este instalat și rulează pe localhost pe port-ul standard (5672). RabbitMQ poate fi descărcat de la link-ul https://www.rabbitmq.com/download.html.

10.3.2.2.Aplicația Hello World

În acest exemplu se vor prezenta două programe în Java:

- un producător care trimite un singur mesaj;
- un consumator care primește mesaje și le afișează.

Mesajul pe care producătorul îl va transmite, va fi *„Hello World”*. Vom da producătorului mesajului (publisher-ul) numele de Send și consumatorului mesajului (receiver) cel de Recv.

Producătorul se va conecta la RabbitMQ, va trimite un singur mesaj, apoi va ieși.

Clasa *Send*, va trebui să importe 3 clase predefinite: *ConnectionFactory*, *Connection* și *Channel* din pachetul com.rabbitmq.client al librăriei rabbitmq. Cu ajutorul acestora, se va crea conexiunea la server-ul de RabbitMQ și se va crea un canal (channel). În exemplul nostru, ne conectăm la un broker de pe mașina locală - prin urmare, prin localhost. Pentru a transmite mesajul mai este necesar să creăm o coadă spre care să-l trimitem. Declararea unei cozi este o operație idempotentă (e.g. coada va fi creată numai dacă aceasta nu există deja). Conținutul mesajului este un vector de octeți, dându-se permisiunea de a transmite orice informație codificată în acest fel.

Consumatorul va rula continuu consumând mesaje de la server-ul de RabbitMQ, spre deosebire de producător, care trimite un singur mesaj. Deasemenea, receiver-ul va afișa fiecare mesaj consumat.

Clasa *Recv*, la fel ca în cazul clasei producătorului, va trebui să importe și ea 3 clase predefinite: *ConnectionFactory*, *Connection* și *Channel* din pachetul com.rabbitmq.client al librăriei rabbitmq. Cu ajutorul acestora, se va crea conexiunea la server-ul de RabbitMQ și se va crea un canal (channel). De asemenea vom declara coada pentru care se va înregistra consumatorul.

Un aspect important îl reprezintă faptul că receiver-ul va trebui să ruleze continuu, pentru a fi pregătit să consume mesajele de pe coada pentru care s-a înregistrat. Ulterior, va trebui să transmitem serverului să trimită mesajele din coadă. Din moment ce ne va transmite mesajele în mod asincron, oferim un callback sub forma unui obiect care va stoca mesajele până când vom fi gata să le folosim. Acesta este roulul subclasei *DeliverCallback*.

Pentru execuția aplicației, veți avea nevoie de librăria *rabbitmq-client.jar*. Deschideți un terminal în locația sursă și rulați următoarele linii de comandă (În Windows, utilizați „**;**” în loc de „**:**” pentru a separa elementele):

- Pentru compilare: javac -cp amqp-client-5.5.1.jar Send.java Recv.java - Rulați producătorul: java -cp .:amqp-client-5.5.1.jar:slf4j-api-1.7.25.jar:slf4j-simple-1.7.25.jar Recv - Rulați consumatorul: java -cp .:amqp-client-5.5.1.jar:slf4j-api-1.7.25.jar:slf4j-simple-1.7.25.jar Send

Consumatorul va afișa mesajul primit de la producător prin intermediul RabbitMQ. Consumatorul va continua să ruleze, așteptând mesajele (Utilizați Ctrl+C pentru a-l opri), deci încercați să rulați producătorul dintr-un alt terminal.

10.3.2.3.Aplicație ce consumă selectiv mesaje

Pentru a asigura selecția mesajelor, așa cum a fost explicat și în secțiunea de fundamentare teoretică, vom folosi o un direct exchanger. Algoritmul de rutare din spatele unui direct exchanger este relativ simplu - se transmite un mesaj către cozile a căror cheie de rutare se potrivește exact cu cheia de rutare a mesajului. Pentru a ilustra acest lucru, luați în considerare următoarea configurație:

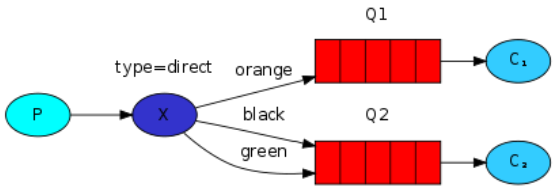


Figura 10.1. Configurație Uni-binding

În această configurație, putem observa direct exchanger-ul **X** care deține binding-uri către două cozi. Prima coadă are un singur binding - cheia de rutare ***orange***. A doua coadă are două binding-uri, o cheie de rutare ***black*** și una ***green***.
Într-o astfel de configurație, un mesaj transmis direct exchanger-ului **X** cu o cheie de rutare ***orange*** va fi direcționat către coada **Q1**. Mesajele cu o cheie de rutare ***black*** sau ***green*** vor fi direcționare către **Q2**. Toate celelalte mesaje vor fi eliminate.

Un alt exemplu sugestiv este acela care face posibil legarea (bindingul) multiplu cu aceeași cheie de rutare, ilustrat în confugrația expusă mai jos:

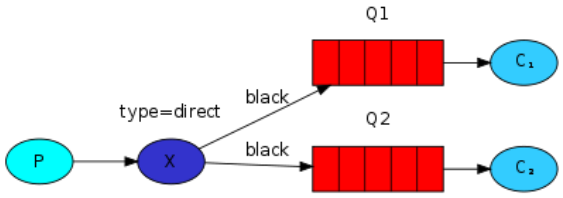


Figura 10.2. Configurație Multi-binding

În acest caz, direct exchanger-ul se va comporta ca un fanout și va transmite mesajul la toate cozile de potrivire. Un mesaj cu cheia de rutare ***black*** va fi livrat atât la **Q1**, cât și la **Q2**.

Vom folosi modelele descrise în secțiunea anterioară pentru un sistem de logare. În loc de fanout vom trimite mesaje către un direct exchanger. Vom utiliza drept cheie de rutare severitatea logging-ului (info, warning, error). Astfel, consumatorii sau aplicațiile care vor recepționa informațiile de logging, vor putea selecta tipul logging-ului primit în funcție de severitatea acestuia.

Pentru emițătorul de log-uri, va trebui să creăm un direct exchange și să trimitem mesajele acestuia. Pe urmă va trebui să creăm binding-uri către fiecare coadă, care vor avea drept cheie de rutare severtatea logging-ului: info, warning, error. Astfel, confugrația finală a aplicației ar trebuie să fie următoarea:

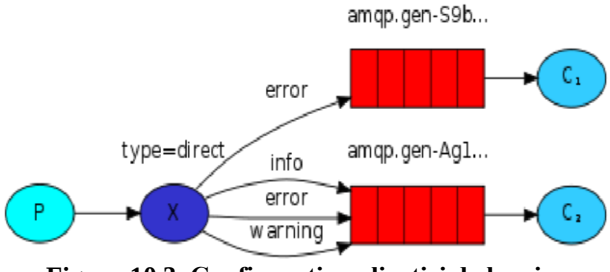


Figura 10.3. Configurația aplicației de logging

Pentru execuția aplicației, veți avea nevoie de librăria **rabbitmq-client.jar**. Deschideți un terminal în locația sursă și rulați următoarele linii de comandă (În Windows, utilizați „**;**” în loc de „**:**” pentru a separa elementele):

- Pentru compilare: javac -cp $CP ReceiveLogsDirect.java EmitLogDirect.java
- Dacă doriți să salvați numai mesajele de warning și de eroare (nu și "info"), deschideți o consolă și rulați comanda: java -cp $CP ReceiveLogsDirect warning error > logs_from_rabbit.log

- Dacă doriți să salvați toate mesajele de logging, deschideți un nou terminal și rulați comanda:
  java -cp $CP ReceiveLogsDirect info warning error
- Pentru a emite un mesaj de eroare, rulați comanda:
  java -cp $CP EmitLogDirect error "Run. Run. Or it will explode."

10.3.2.4.Aplicație ce consumă mesaje pe baza unui pattern (Topic)

Mesajele trimise către un **topic exchange** nu pot să aibă un **routing_key** (cheie de rutare) arbitrar – trebuie să fie o listă de cuvinte, delimitate de puncte. Cuvintele pot să fie orice, dar de obicei ele specifică unele caracteristici legate de mesaj. Câteva exemple de chei de rutare valide: **stock.usd.nyse**, **nyse.vmw**, **quick.orange.rabbit**. Pot exista oricâte cuvinte în cheile de rutare, limitate însă de dimensiunea de 255 de octeți.

Cheia de legare trebuie să fie, de asemenea, în aceeași formă. Logica din spatele unui topic exchange este similară cu cea de la **direct exchange** – un mesaj trimis cu o anumită cheie de rutare va fi livrat către toate cozile care sunt legate cu o cheie de legare corespunzătoare. Cu toate acestea, există două cazuri importante și speciale pentru cheile de legare:

- ***** (stea) poate substitui exact un cuvânt;
- # (hash) poate substitui zero sau mai mult cuvinte.

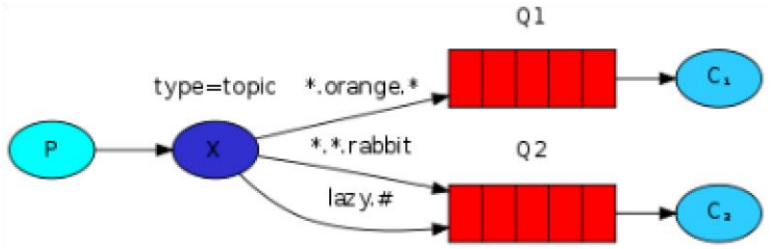


Figura 10.4. Configurație aplicație ce consumă selectiv mesaje(topicuri)

În acest exemplu, vom trimite mesaje care descriu toate animalele. Mesajele vor fi trimise cu o cheie de rutare care constă din trei cuvinte (deci vor conține două puncte). Primul cuvânt din cheia de rutare va descrie viteza, al doilea o culoare și al treilea o specie: <viteză>.<culoare>.<specie>.

S-au creat trei legături: Q1 este legat cu cheie de legare *.orange.* și Q2 cu *.*.rabbit și lazy.#. Acest legături pot fi sumarizate astfel:

- Q1 este interesat de toate animalele portocalii (*orange*);
- Q2 vrea să știe totul despre iepuri (*rabbits*) și totul despre animalele leneșe (*lazy animals*).

Un mesaj cu o cheie de rutare quick.orange.rabbit va fi livrat la ambele cozi, asemenea mesajului lazy.orange.elephant. Pe de altă parte, quick.orange.fox va merge doar la prima coadă, în timp ce lazy.brown.fox va fi livrat către cea de-a doua. Mesajul lazy.pink.rabbit va fi livrat către a doua coadă doar o dată, deși se potrivește cu două legături, în timp ce quick.brown.fox nu se potrivește cu nicio legătură deci va fi șters.

În cazul în care se trimite un mesaj care nu respectă contractul (adică cu unul sau patru cuvinte), precum orange sau quick.orange.male.rabbit, acestea nu se vor potrivi niciunei legături și vor fi pierdute. Pe de altă parte lazy.orange.male.rabbit, chiar dacă are patru cuvinte, se potrivește cu ultima legătură și va fi trimisă către cea de-a doua coadă.

Astfel, atunci când o coadă este legată cu o cheie # (hash), ea va primi toate mesajele, indiferent de cheia de rutare – ca în fanout exchange. Când caracterele speciale * (stea) și # (hash) nu sunt folosite în legături, topic exchange se va comporta ca direct exchange.

În cadrul exemplelor se va folosi un topic exchange pentru un sistem de logare folosindu-ne de ipoteza conform căreia cheile de rutare a log-urilor vor conține două cuvinte: <facility>.<severity>. Codul pentru rularea exemplelor se găsește în fișierele EmitLogTipic.java, respectiv ReceiveLogsTopic.java. E nevoie ca acestea să fie compilate în primul rând, iar rularea acestora se poate face din linia de comandă folosind exemplele de comenzi prezentate mai jos.

- Pentru compilare: javac -cp $CP ReceiveLogsTopic.java EmitLogTopic.java - Pentru primirea tuturor log-urilor: java -cp $CP ReceiveLogsTopic "#"
- Pentru primirea tuturor log-urilor din kern: java -cp $CP ReceiveLogsTopic "kern.*"
- Sau dacă doriți să aflați doar despre log-urile de nivel criticial: java -cp $CP ReceiveLogsTopic "*.critical" - Puteți crea legături multiple: java -cp $CP ReceiveLogsTopic "kern.*" "*.critical"
- Și pentru a emite un log cu o cheie de rutare de tipul kern.critical:
  java -cp $CP EmitLogTopic "kern.critical" "A critical kernel error"

10.4.Întrebări teoretice

10.4.1. Definiți conceptul de broker de mesaje.

10.4.2. Care sunt tipurile principale de interacțiuni ce pot fi implementate folosind protocolul AMQP și prin ce sunte acestea realizate ?

10.4.3.Explicați și argumentați diferențele conceptuale între coadă și canal. Explicați modelul de mesageriepub-sub și particularitățile sale în tehnologia RabitMQ

10.5.Probleme propuse

10.5.1.Construiți o aplicație în care producătorul va trimite mesaje text către consumatori. Un consumator va afișa mesajul nemodificat, altul îl va afișa în ordine inversă, etc. Implementați o soluție a aplicației folosind:

a. Trei consumatori și o singură coadă;
b. Patru consumatori, trei cozi și doi exchanger-i;
c. Un consumator și o coadă. Mesajele vor fi direcționate către coadă doar dacă conținutul lor va fi identic cu textul „RabbitMQ”. Implementarea nu va conține secvențe de cod if-then-else;
d. Trei consumatori și două cozi. Prima coadă va primi mesaje care nu conțin cifre, iar cea de-a doua, mesaje care conțin cel puțin o cifră. Implementarea nu va trebui să conțină secvențe de cod if-then-else.

10.5.2.Implementați problema anterioară folosind tehnologii cloud (CloudAMQP – vezi aplicația prezentată în laborator).

10.6.Referințe bibliografice

1. Documentația oficială CloudAMQP- https://www.cloudamqp.com
2. Documentația oficială RabbitMQ -: https://www.rabbitmq.com

- **Executor**: proces responsabil pentru executarea unui task. Numărul de executori folosiți în programe este direct legat de cantitatea de timp în care task-ul unui job este executat. Figura 12.2 arată modul de folosire a executorilor la nivelul unei arhitecturi. Există însă un prag dincolo de care creșterea numărului de executori nu mai reduce timpul de execuție.
- **Driver**: Programul/procesul responsabil pentru rularea job-ului peste motorul Spark
- **Master**: Mașina pe care rulează programul Driver
- **Slave**: Mașina pe care rulează programul Executor
- **Stages**: Job-urile sunt împărțite în etape (stages). Etapele sunt clasificate ca etape Map sau Reduce.

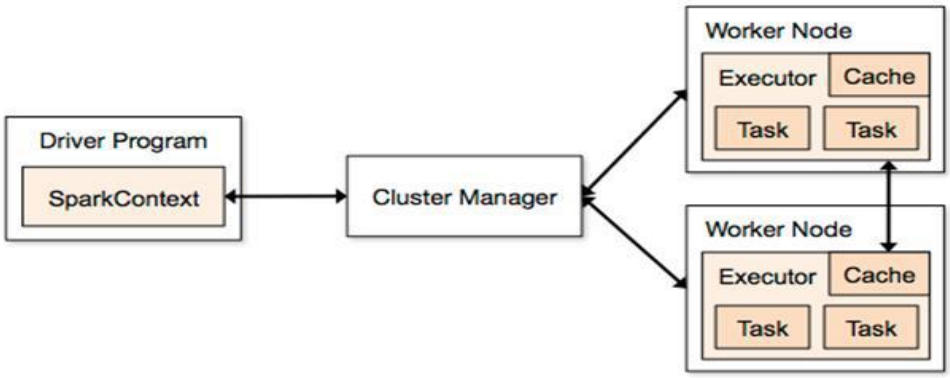


Figura 11.2. Cluster Spark [1]

11.2.1.Resilient Distributed Datasets (RDD)

Modelul MapReduce clasic are limitări în ceea ce privește aplicațiile care aplică iterativ o anumită funcție pe aceeași sursă de date la fiecare pas de procesare, cum ar fi aplicațiile de machine learning sau de procesare a grafurilor ,aplicații ce necesită un model de procesare iterativ. Modelul MapReduce oferă timpi de execuție mari pentru aplicații iterative deoarece el stochează datele pe hard disk-uri și reîncarcă aceleași date direct de pe hard disk la fiecare pas. Sarcinile (jobs) MapReduce se bazează pe replicarea fișierelor în sistemul distribuit de fișiere pentru a implementa mecanisme de recuperarea erorilor (fault recovery). Acest mecanism adaugă supraîncărcare semnificativă la transmisia în rețea atunci când e vorba de replicarea unor fișiere de dimensiune mare.

Pentru înlăturarea acestor limitări, în Spark au fost implementate mecanisme speciale de optimizare a modelului. Astfel , RDD este o memorie abstractă tolerantă la erori care evită replicarea datelor și minimizează căutarea pe disk. Aceasta permite aplicațiilor să stocheze datele în memorie în diferite etape de procesare, ceea ce duce la o accelerare substanțială a reutilizării viitoare a datelor. În plus, RDD-urile reține operațiile folosite pentru a le putea reconstrui, astfel atunci când are loc o eroare, RDD-urile pot reconstrui datele cu supraîncărcare minimă a rețelei (datorată mecanismelor de transfer necesare reconstituirii lor).

RDD-urile oferă unele restricții privind folosirea memoriei partajate pentru a permite toleranța scăzută a erorilor de supraîncărcare, astfel, acestea sunt read-only și partiționate. Fiecare partiție conține înregistrări care pot fi create prin operații deterministe numite transformări.

Transformările includ operații de tipul *map, reduce, groupBy și join*. RDD-urile pot fi create prin alte RDD-uri existente sau printr-un set de date într-un spațiu de stocare stabil. Aceste restricții facilitează reconstrucția partițiilor pierdute deoarece fiecare RDD are suficiente informații din alte RDD-uri despre cum să le reconstruiască. Când există o cerere de păstrare (cache) a RDD-ului, motorul de procesare Spark stochează partițiile implicit în memorie. Cu toate acestea, partițiile pot fi stocate pe hard disk atunci când nu este disponibil spațiu suficient de memorie sau utilizatorii solicită memorarea (cache) pe hard disk. RDD-urile conțin date grupate pe baza unei chei asociate fiecărei înregistrări, după care acestea sunt distribuite în mod corespunzător.

Fiecare RDD constă într-un set de partiții, un set de dependințe cu RDD-ul părinte, o funcție de calcul și metadate despre schema de partiționare. Funcția definește modul în care un RDD este construit din RDD-ul părinte. De exemplu, dacă un RDD este reprezentat printr-un fișier HDFS (sistem d efișiere

Hadoop), atunci fiecare partiție reprezintă câte un bloc al fișierului HDFS. Metadatele vor conține informații despre fișierul HDFS, cum ar fi locația și numărul de blocuri. Acest RDD este creat dintr-un spațiu de stocare stabil, deci nu are dependințe cu RDD-urile părinte. O transformare poate fi efectuată pe unele RDD-uri, care reprezintă diferite fișiere HDFS, iar output-ul transfornării este un nou RDD, ce va conține un set de partiții în funcție de natura transformării. Fiecare partiție va depinde de un set de alte partiții din RDD-ul părinte. Partițiile părinte sunt folosite pentru a recompune partiția fiu atunci când este necesar.

Frameworkul Spark clasifică dependințele cu RDD-ul părinte în două tipuri: înguste (narrow) și largi (wide). Dependințele înguste indică faptul că fiecare partiție fiu RDD depinde de un număr fix de partiții din părintele RDD, cum ar fi transformările map. Dependințele largi indică faptul că fiecare partiție fiu RDD poate depinde de toate partițiile părintelui RDD, cum ar fi transformările groupByKey. Această clasificare ajută Spark să îmbunătățească mecanismul de execuție și recuperare. RDD-urile cu dependințe înguste pot fi compuse într-un singur cluster, cum ar fi operația map urmată de operația filter. În mod contrariu, depedințele largi necesită ca datele din partițiile părinte să fie amestecate între diferite noduri. Mai mult, o eroare a nodului poate fi recuperată mai eficient cu o dependință îngustă, deoarece Spark va recompune un număr fix de partiții pierdute, iar aceste partiții pot fi recompuse în paralel pe noduri diferite. În schimb, un singur eșec al nodului ar putea necesita o execuție completă în cazul dependințele largi. Această clasificare asigura ca frameworkul Spark să programeze planul de execuție și să genereze checkpoint-uri pentru procesul de recovery.

Planificatorul de sarcini al Spark-ului folosește structura fiecărui RDD pentru a optimiza planul de execuție. Obiectivul său este de a construi un graf aciclic direct (DAG - Directed Acyclic Graph) al etapelor de calcul pentru fiecare RDD. Fiecare etapă conține cât mai multe transformări posibile cu dependințe înguste. O nouă etapă începe atunci când există o transformare cu dependințe largi sau o partiție care este stocată în alte noduri. Planificatorul plasează sarcini bazate pe localizarea datelor pentru a minimiza comunicarea în rețea. Dacă o sarcină are nevoie să proceseze o partiție din cache, planificatorul trimite această sarcină unui nod care are partiția în cache.

11.2.2. Operații Spark

Transformări

Spark Transformation este o funcție care produce un RDD nou din RDD-uri existente. Primește ca input unul sau mai multe RDD-uri și returnează unul sau mai multe RDD-uri. De fiecare dată când are loc o transformare se produce un nou RDD. Ca o regulă generală, RDD-urile nu pot fi modificate pentru că sunt imutabile.

Transformările sunt lente în general și sunt executate atunci când se face apel la o acțiune, dar nu sunt executate întotdeauna imediat. După transformare, un nou RDD este creat care poate fi mai mic (transformări precum: filer, count, distinct, sample) sau mai mare(transformări precum: flatMap(), union(), cartesian()) sau de aceași dimensiune cu sursa.
Cele două tipuri de transformări sunt:

- *Narrow transformation* - toate elementele care sunt necesare pentru a calcula înregistrările în partiția unică trăiesc în partiția unică a RDD-ului părinte. Un subset limitat de partiții este folosit pentru a calcula rezultatul. Narrow transformation-urile sunt rezultatul transformărilor map(), filter().
- *Wide transformation* - toate elementele necesare pentru a calcula înregistrările din partiția unică pot exista în mai multe partiții ale RDD părinte. Wide transformation-urile sunt rezultatul transformărilor grupuluiKey () și reducebyKey ().

Câteva exemple de transformări în Spark: map(func), flatMap(), filter(func), mapPartitions(func), mapPartitionWithIndex(), union(dataset), instersection(other-dataset), distinct(), groupBykey(), reduceByKey(func, [numTasks]), sortByKey(), join(), coalesce().

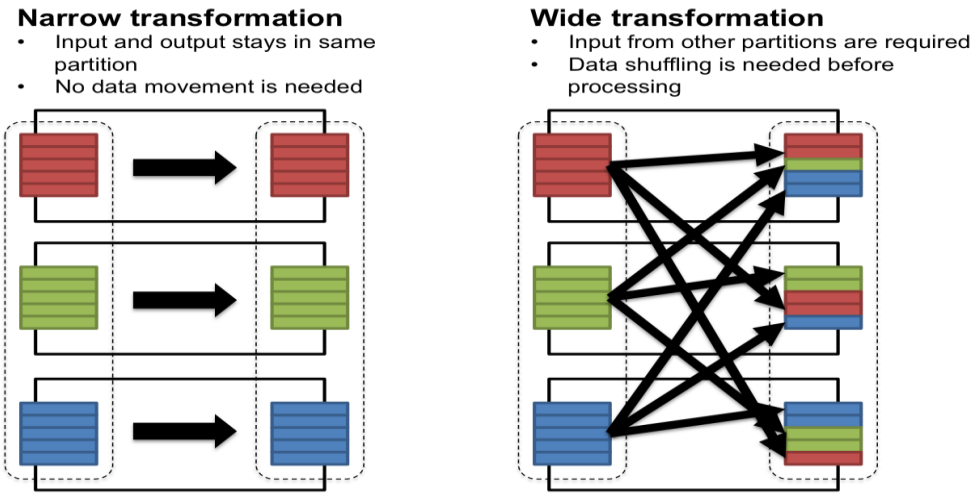


Figura 11.3. Transformări Narrow vs. Wide [2]

Acțiuni

Transformările permit construirea unui plan specific de transformare logică. Pentru a declanșa calculul, se execută o acțiune. Acțiunea instruiește frameworkul să calculeze rezultatul dintr-o serie de transformări. Cea mai simplă acțiune este numărul care dă numărul total de înregistrări din DataFrame. Acțiunea este una din modalitățile de transmitere a datelor de la Executor la driver.

Câteva exemple de acțiuni în spark: *count(), collect(), take(n), top(), reduce(), fold(), aggregate(), foreach().*

11.2.3. Librării Spark

Spark Streaming

Apache Spark Streaming este o extensie Apache Spark API și este capabil să proceseze stream-uri de date în timp real. Apache Spark este capabil de a prelua cantități uriașe de date de intrare, încărcându-le în mai multe noduri cluster și procesându-le în paralel. Datele sunt separate în seturi mai mici și imutabile și asignate către diverse noduri din sistem. Fiecare nod cluster procesează doar datele ce i-au fost asignate ,astfel asigurându-se o procesare rapidă folosindu-se conceptul de localitate a datelor. Aplicațiile Spark sunt executate în paralel (mai multe instanțe ale aceluiași proces funcționează pe diferite seturi de date pe fiecare nod din cluster). Pentru a aloca resurse și pentru procesarea nodurilor se folosesc manageri la nivelul cluster-ului. Odată ce nodurile sunt alocate, motorul Spark accesează executorii din noduri, ce reprezintă procese ce primesc, stochează și procesează datele dintr-un anumit nod.

Avantajul folosirii unui astfel de framework este acela că fiecare nod are propriul proces de execuție care rulează cât timp aplicația Spark nu e oprită, rulând task-uri multiple (folosind thread-uri multiple). Totodată, Spark nu impune utilizarea managerului său autonom de cluster și sprijină utilizarea managerilor de clustere (cum ar fi Mesos sau YARN), capabili să se conecteze la mai multe aplicații.

Programele Apache Spark pot fi scrise în diverse limbaje de programare, datorită faptului că se oferă API pentru:Python,Java,Scala. Complementar, ca extensii la aplicația de bază Apache Spark, sunt disponibile, de asemenea:

- execuția interogărilor SQL;
- procesarea operațiilor Map și Reduce;
- machine learning;
- prelucrarea datelor grafice și procesarea fluxului de date.

Figura 11.4.Arhitectura Spark Streaming

Apache Spark Streaming permite integrarea diferitelor sisteme de mesagerie și introducere de date, cum ar fi Amazon Kinesis, Apache Flume, Apache Kafka, sisteme de social media cum ar fi Twitter, sisteme de baze de date distribuite precum Hadoop Distributed File System pentru a accepta datele de intrare pentru procesare. Datele o dată introduse în sistem sunt prelucrate utilizând funcții de nivel înalt, cum ar fi: map, reduce, window și join. În cele din urmă, rezultatul generat poate fi trimis la mai multe data sinks, conform cerințelor utilizatorului. De exemplu, datele de ieșire generate pot fi stocate în sisteme de fișiere distribuite, cum ar fi HDFS sau pot fi stocate în sisteme de baze de date .

Stream-urile de date sunt acceptate în sistemul Apache Spark Streaming de la o sursă externă de date. Aceste stream-uri continue de date sunt în continuare împărțite în mai multe secțiuni sau batch-uri, care sunt apoi trimise la motorul Apache Spark. Acesta tratează aceste batch-uri de date ca seturi de date imutabile care trebuie procesate conform logicii aplicației. Datele de ieșire generate după prelucrare sunt de asemenea livrate în batch-uri.

Sream-urile de date continue de intrare sunt reprezentate ca fluxuri discreționate (Discretised Streams) sau DStreams în Apache Spark Streaming. Aceasta este o abstracție la nivel înalt furnizată pentru a reprezenta stream-urile continue primite ca intrări sau stream-urile de date generate de procesarea stream-urilor de intrare. Sistemul DStream este structurat ca o serie continuă de seturi de date imutabile, distribuite și partiționate, numite Seturi de date distribuite reziliente (Resilient Distributed Data sets-RDD) . Fiecare RDD aparține unui anumit Dstream și conține date dintr-un anumit interval de timp. Acest lucru este reprezentat de figura următoare:

Figura 11.5.Reprezentarea modului in care DStream-urile sunt partiționate in Resilient Distributed Datasets

Caracteristici Apache Spark Streaming:

- *Fault tolerance*- datorită RDD ce reprezintă sunt seturi de elemente de date imutabile care pot fi re-calculate în cazul unui eșec.
- *Throughput* - pentru a asigura o performanță ridicată, Spark oferă facilitatea de a primi mai multe stream-uri în paralel. Aceasta se poate realiza dacă sunt create mai multe receptoare ce acceptă simultan date de la DStreams de intrare pentru a menține o rată ridicată de ingestie a datelor.
- *Scalabilitate* - fiind un framework bazat pe cluster, Apache Spark Streaming acceptă executarea job-urilor în mai multe noduri conectate unul la celălalt. Aceste noduri cluster funcționează în paralel pentru a echilibra sarcina de procesare. Apache Spark este capabil să crească sau să scadă dinamic numărul de noduri care lucrează în paralel utilizând metoda Dynamic Allocation. În funcție de statusul batch job -urilor, Apache Spark include sau exclude nodurile procesoarelor, făcându-l astfel foarte scalabil.

Spark ML (Machine Learning Library)

Machine Learning Library (MlLib) este biblioteca Spark a funcțiilor de machine learning. Conceput pentru a funcționa în paralel pe clustere, MLlib conține o varietate de algoritmi de machine learning și este accesibil din diverse limbajele de programare. Designul și arhitectura MlLib sunt simple: sunt permise invocări ale diferiților algoritmi pe seturi de date distribuite, reprezentând aceste date ca RDD-uri. Acest modul oferă funcționalități care includ: statistici, clasificare, regresie, filtrare colaborativă, clusterizare, extracție de caracteristici, optimizare, etc.

Spark SQL

Spark SQL este o componentă peste Spark Core care oferă suport pentru acces la date structurate și nestructurate. Acesta oferă o interfață care interacționează cu Spark prin intermediul limbajelor de interogare precum SQL și HiveQL. Interogările sunt traduse în operații Spark iar tabelele bazei de date sunt reprezentate ca RDD-uri.

Din Spark versiunea 1.3 data frame-urile au fost introduse în Apache Spark, astfel încât datele Spark să poată fi procesate într-o formă tabulară și funcțiile tabulare (cum ar fi selecție, filtrare, groupBy) pot fi folosite pentru procesarea datelor. Modulul Spark SQL se integrează cu formatele Parquet și JSON pentru a permite stocarea datelor în formate care reprezintă în cea mai adecvată manieră pentru aplicație , datele, fapt ce oferă de asemenea, mai multe opțiuni de integrare cu sistemele externe.

Spark GraphX

GraphX este o nouă componentă în Spark pentru grafuri și pentru calculul paralel cu grafuri. La un nivel înalt, GraphX extinde Spark RDD introducând o nouă abstractizare grafică: un multigraf direcționat cu proprietăți atașate la fiecare vârf și muchie. Pentru a susține diverse operații pe grafuri, GraphX expune un set de operatori fundamentali (de exemplu, subgraf, joinVertices și agregateMessages), precum și o variantă optimizată a API-ului Pregel. În plus, GraphX include o colecție tot mai mare de algoritmi și constructori pentru a simplifica sarcinile de analiză grafică.

11.2.4. Principalele funcții Spark: transformări, operatori, acțiuni

Cele mai importante transformări, operatori și actiuni sunt prezentate în tabelele 11.1, 11.2, 11.3 , preluate din [6**].**

Tabel 11.1.Transformări uzuale(din [6])

Transformarea și scopul	**Exemplu și Rezultat**
filter(func) Purpose: new RDD by selecting those data elements on which func returns true	scala> val rdd = sc.parallelize(List("ABC","BCD","DEF")) scala> val filtered = rdd.filter(_.contains("C")) scala> filtered.collect() Result: Array[String] = Array(ABC, BCD)
map(func) Purpose: return new RDD by applying func on each data element	scala> val rdd=sc.parallelize(List(1,2,3,4,5)) scala> val times2 = rdd.map(_*2) scala> times2.collect() Result: Array[Int] = Array(2, 4, 6, 8, 10)
flatMap(func) Purpose: Similar to map but func returns a Seq instead of a value. For example, mapping a sentence into a Seq of words	scala> val rdd=sc.parallelize(List("Spark is awesome","It is fun")) scala> val fm=rdd.flatMap(str=>str.split(" ")) scala> fm.collect() Result: Array[String] = Array(Spark, is, awesome, It, is, fun)

reduceByKey(func,[numTasks]) Purpose: : To aggregate values of a key using a function. “numTasks” is an optional parameter to specify number of reduce tasks	scala> val word1=fm.map(word=>(word,1)) scala> val wrdCnt=word1.reduceByKey(_+_) scala> wrdCnt.collect() Result: Array[(String, Int)] = Array((is,2), (It,1), (awesome,1), (Spark,1), (fun,1))
groupByKey([numTasks]) Purpose: To convert (K,V) to (K,Iterable<V>)	scala> val cntWrd = wrdCnt.map{case (word, count) => (count, word)} scala> cntWrd.groupByKey().collect() Result: Array[(Int, Iterable[String])] = Array((1,ArrayBuffer(It, awesome, Spark, fun)), (2,ArrayBuffer(is)))
distinct([numTasks]) Purpose: Eliminate duplicates from RDD	scala> fm.distinct().collect() Result: Array[String] = Array(is, It, awesome, Spark, fun)

Tabel 11.2.Operatori uzuali pe seturi de date (din [6])

Transformarea și scopul	**Exemplu și Rezultat**
union() Purpose: new RDD containing all elements from source RDD and argument.	Scala> val rdd1=sc.parallelize(List(‘A’,’B’)) scala> val rdd2=sc.parallelize(List(‘B’,’C’)) scala> rdd1.union(rdd2).collect() Result: Array[Char] = Array(A, B, B, C)
intersection() Purpose: new RDD containing all elements from source RDD and argument.	Scala> rdd1.intersection(rdd2).collect() Result: Array[Char] = Array(B
cartesian() Purpose: new RDD cross product of all elements from source RDD and argument.	Scala> rdd1.cartesian(rdd2).collect() Result: Array[(Char, Char)] = Array((A,B), (A,C), (B,B), (B,C))
subtract() Purpose: new RDD created by removing data elements in source RDD in common with argument	scala> rdd1.subtract(rdd2).collect() Result: Array[Char] = Array(A)
join(RDD,[numTasks]) Purpose: When invoked on (K,V) and (K,W), this operation	scala> val personFruit = sc.parallelize(Seq((“Andy”, “Apple”), (“Bob”, “Banana”), (“Charlie”, “Cherry”), (“Andy”,”Apricot”))) scala> val personSE = sc.parallelize(Seq((“Andy”,

creates a new RDD of (K, (V,W))	"Google"), ("Bob", "Bing"), ("Charlie", "Yahoo"), ("Bob","AltaVista"))) scala> personFruit.join(personSE).collect() Result: Array[(String, (String, String))] = Array((Andy,(Apple,Google)), (Andy,(Apricot,Google)), (Charlie,(Cherry,Yahoo)), (Bob,(Banana,Bing)), (Bob,(Banana,AltaVista)))
cogroup(RDD,[numTasks]) Purpose: To convert (K,V) to (K,Iterable<V>)	scala> personFruit.cogroup(personSE).collect() Result: Array[(String, (Iterable[String], Iterable[String]))] = Array((Andy,(ArrayBuffer(Apple, Apricot),ArrayBuffer(Google))), (Charlie,(ArrayBuffer(Cherry),ArrayBuffer(Yahoo))), (Bob,(ArrayBuffer(Banana),ArrayBuffer(Bing, AltaVista))))

Tabel 11.3. Acțiuni uzuale (din [6])

Acțiunea și scopul	**Exemplu și Rezultat**
count() Purpose: Get the number of data elements in the RDD	scala> val rdd = sc.parallelize(List('A','B','C')) scala> rdd.count() Result: Long = 3
collect() Purpose: get all the data elements in an RDD as an array	scala> val rdd = sc.parallelize(List('A','B','C')) scala> rdd.collect() Result: Array[Char] = Array(A, B, C)
reduce(func) Purpose: Aggregate the data elements in an RDD using this function which takes two arguments and returns one	scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.reduce(_+_) Result: Int = 10
take (n) Purpose: : fetch first n data elements in an RDD. Computed by driver program.	Scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.take(2) Result: Array[Int] = Array(1, 2)
foreach(func) Purpose: execute function for each data element in RDD. Usually used to update an accumulator(discussed later) or interacting with external systems.	Scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.foreach(x=>println("%s*10=%s". format(x,x*10))) Result: 1*10=10 4*10=40 3*10=30 2*10=20
first()	scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.first()

Purpose: retrieves the first data element in RDD. Similar to take(1)	Result: Int = 1
saveAsTextFile(path) Purpose: Writes the content of RDD to a text file or a set of text files to local file system/ HDFS	scala> val hamlet = sc.textFile("/Users/akuntamukkala/ temp/gutenburg.txt") scala> hamlet.filter(_.contains("Shakespeare")). saveAsTextFile("/Users/akuntamukkala/temp/filtered") Result: akuntamukkala@localhost~/temp/filtered$ ls _SUCCESS part-00000 part-00001

11.2.5. Instalare și configurare

Pentru a putea fi utilizat frameworkul Spark, este necesară instalarea și configurarea prealabilă a unor librării, după cum urmează**:**

1. Se instalează Scala:

- Descarcă kitul de la: http://downloads.lightbend.com/scala/2.11.8/scala2.11.8.msi
- Setează environmental variables:
  - User variable:
    - Variable: SCALA_HOME;
    - Value: C:\Program Files (x86)\scala
  - ii. System variable:
    - Variable: PATH
    - Value: C:\Program Files (x86)\scala\bin

2. Se *instalează Java 8:*

- Descarcă Java 8 de la: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Setează environmental variables:
  - User variable:
    - Variable: JAVA_HOME
    - Value: C:\Program Files\Java\jdk1.8.0_91
  - System variable:
    - Variable: PATH
    - Value: C:\Program Files\Java\jdk1.8.0_91\bin

3. Se *instalează Eclipse sau Intelij IDEA sau orice alt IDE preferat.*
4. Se *instalează Spark 1.6.1*.

- Descarcă kitul de la: http://spark.apache.org/downloads.html și dezarhivează-l pe partiția D, de exemplu D:\Spark.
- Setează environmental variables:
  - User variable:
    - Variable: SPARK_HOME
    - Value: D:\spark\spark-1.6.1-bin-hadoop2.6
  - System variable:
    - Variable: PATH
    - Value: D:\spark\spark-1.6.1-bin-hadoop2.6\bin

5. Se descarcă Windows Utilities.

- Descarcă kitul de la:https://github.com/steveloughran/winutils/tree/master/hadoop-2.6.0/bin
- Pune-l în folderul D:\spark\spark-1.6.1-bin-hadoop2.6\bin

6. Se execută Spark în cmd pentru verificare:

- "spark-shell din directorul de spark"

7. Se instalează Maven 3.3.

- Descarcă Apache-Maven-3.3.9 de la: http://apache.mivzakim.net/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-bin.zip și dezarhivează-l pe disk-ul D, de exemplu: D:\apache-maven-3.3.9 a.

- Set Environmental variables:
  - User variable
    - Variable: MAVEN_HOME
    - Value: D:\apache-maven-3.3.9
  - System variable
    - Variable: Path
    - Value: D:\apache-maven-3.3.9\bin

Detalii suplimentare: http://www.ics.uci.edu/~shantas/Install_Spark_on_Windows11.pdf

11.3.Exemple

11.3.1.PageRank

Programul calculează PageRank-ul pentru URL-urile din setul de date folosit ca input. Fiecare linie conține câte un URL care e separat prin spațiu de câte un vecin de al său. Un exemplu descriptiv generic pentru structura fișierului de intrare este cel de mai jos:

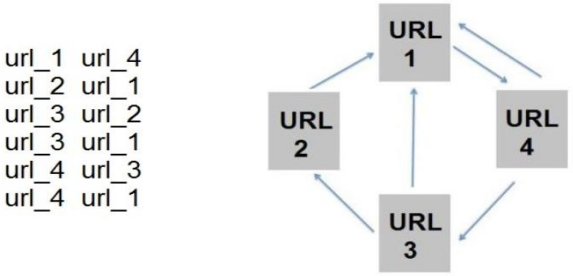


Figura 11.8. Exemplu page rank

Descrierea algoritmului:

Acest algoritm afișează o distribuție de probabilități ,ce reprezintă probabilitatea ca o persoană care dă click la întâmplare pe legături web să ajungă la o anumită pagină web.
Dacă se rulează PageRank pe exemplul anterior se obțin următoarele date:

```
url_4 has rank: 1.3705281840649928.
url_2 has rank: 0.4613200524321036.
url_3 has rank: 0.7323900229505396.
url_1 has rank: 1.4357617405523626.
```

Rezultatul indică faptul că URL_1 are cel mai mare page rank, acesta fiind urmat de URL_4, apoi URL_3 si URL_2.

Algoritmul lucrează în următorul mod:

- Dacă un URL (pagina) e cel mai referit de celelalte URL-uri atunci rank-ul crește, deoarece a fi referit înseamnă că e important (exemplu, URL_1)
- Dacă un URL important (exemplu, URL_1) referă alte URL-uri (exemplu, URL_4) aceasta va crește rank-ul URL-ului destinație.

Astfel că e de înțeles de ce URL_4 urmează după URL_1, și în același timp din graficul de mai sus se înțelege că URL_2 are cel mai mic rank al paginii deoarece e referit cel mai puțin.

Programul care calculează page rank, este format din trei părți principale: prima parte citește fișierul de intrare urmând ca în partea a doua fiecărui URL să i se atașeze o valoare (rank), iar ultima parte conține bucla principală care calculează rank-urile.

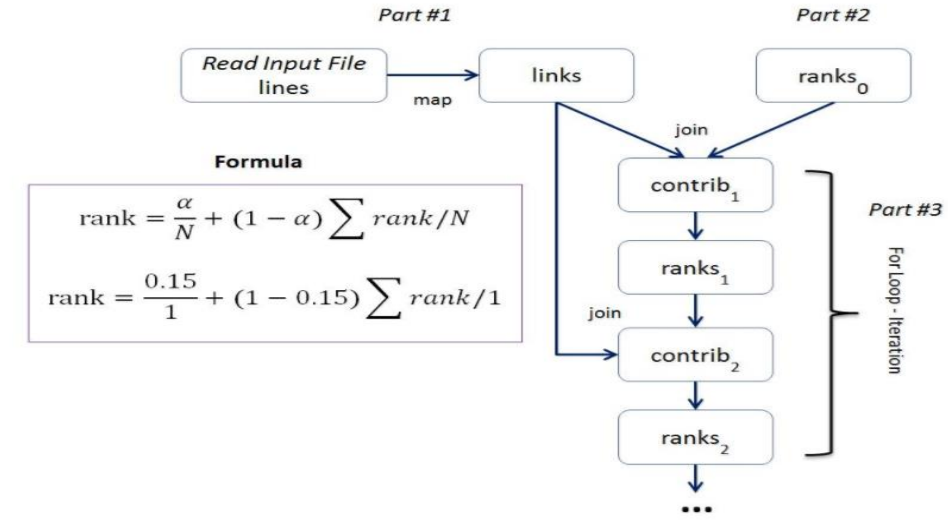


Figura 11.9. Modelul de execuție PageRank

11.3.2.WordCount

Programul dat ca exemplu calculează numărul de cuvinte dintr-un fișier text dat ca input. Orice fișier care conține text poate fi utilizat ca fișierului de intrare.

Pentru a putea număra câte cuvinte conține un fișier text e necesar în primul rând să se creeze un obiect de tip SparkSession care reprezintă o conexiune la un cluster Spark și poate fi folosit pentru a crea RDD-uri (Resilient Distributed Dataset) și alte tipuri de variabile.

```
SparkSession spark = SparkSession
        .builder()
        .appName("JavaWordCount")
        .config("spark.master", "local")
        .getOrCreate();
```

Pasul următor este să se creeze un RDD (Resilient Distributed Dataset) din fișierul de intrare care va permite ca ulterior sa se aplice diferite funcții pentru prelucrarea fișierului și obținerea rezultatelor.

Divizarea textului în cuvinte separate se va face cu transformarea flatMap(func) care va returna un RDD nou care conține cuvintele din text.

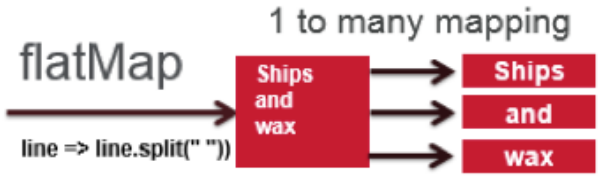


Figura 11.10. Transformarea flatMap

Cuvintele astfel obținute se vor transforma în perechi de tip (cheie, valoare) folosind transformarea mapPair(func).

Pentru a număra de câte ori apare un cuvânt se vor combina valorile care au aceeași cheie din perechile obținute anterior folosind transformarea reducebyKye(func). Se va returna un RDD care conține pereche (cuvânt, număr de apariții).

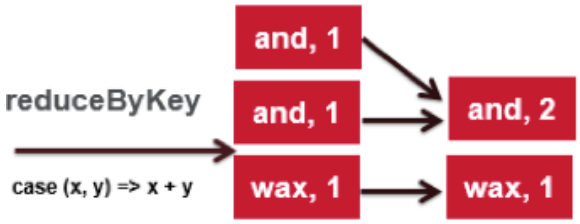


Figura 11.11.Transformarea reduceByKey

Pentru a obține perechea de tuple și a o putea folosi ulterior se va folosi acțiunea collect() care va returna elementele setului într-un array.

11.3.3.SVM metoda stochastică

Algoritmul reflect[modul]n care se antrenează un Support Vector Machine (SVM) folosind Stochastic Gradient Descent (SGD). SVMs sunt modele de învățare supervizată cu algoritmi de învățare asociați care analizează datele utilizate pentru analiza de clasificare și regresie. Se dă un set de exemple de antrenare, fiecare fiind marcat ca aparținând uneia din cele două categorii, un algoritm SVM va construi un model care asignează noile modele la una dintre cele două categorii.

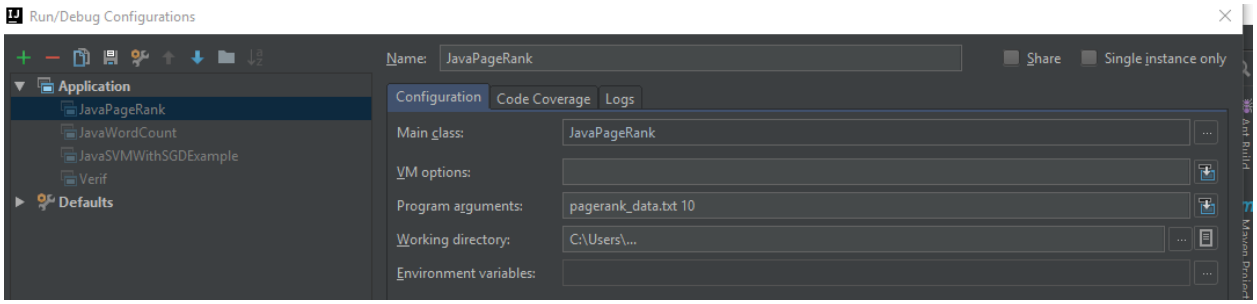
Stochastic gradient descent (SGD) este o aproximare randomizată a algoritmului de gradient descent (batch) pentru a minimiza o funcție obiectiv diferențiată continuă. În învățarea automată supravegheată, funcția obiectiv este o funcție de pierdere - loss function(logistică, suma pătratelor, etc.). În tehnica ML (machine learning), metodele liniare utilizează metode convexe de optimizare pentru a optimiza funcțiile obiectiv, iar în cazul de față SVM folosește SGD.

Pentru a putea clasifica anumite date, în primul rând să va crea un obiect de tip SparkConf ce se transmite ca parametru la crearea unui nou obiect de tip SparkContext care reprezintă o conexiune la un cluster Spark și poate fi folosit pentru a crea RDD-uri (Resilient Distributed Dataset) și alte tipuri de variabile.

```
SparkConf conf = new SparkConf()
        .setAppName("JavaSVMWithSGDExample")
        .setMaster("local");
SparkContext sc = new SparkContext(conf);
```

Pasul următor este să se creeze un RDD (Resilient Distributed Dataset) din fișierul de intrare care va permite ca ulterior sa se aplice diferite funcții pentru prelucrarea fișierului și obținerea rezultatelor. Pentru a putea aplica SVM sunt necesare date de test și de training, astfel că RDD-ul inițial este împărțit în două RDD-uri generând astfel 60% date pentru training și 40% date pentru test. Pasul următor reprezintă aplicarea funcției train() din librăria SVMWithSGD care rulează algoritmul de training pentru a construi modelul. Ca pași finali se obțin rezultatele pe setul de test și se preiau metricile de evaluare urmând ca modelul să fie salvat.

Elemente de analiză și evaluare. Fiecare dintre cele trei programe necesită unul sau mai mulți parametrii pentru rulare. Pentru a seta acești parametrii în mediul de utilizare IntelliJ e necesar să se selecteze din meniu Run-> Edit Configuration și se va obține următorul rezultat



Următorul pas este să se selecteze simbolul  din partea stânga sus și se selectează din drop down-ul apărut *Applicaton* și încep să se completeze câmpurile Name – cu un nume dorit pentru a ști clasa care se rulează, Main Class – numele clasei, Program arguments – trebuie specificate argumentele programului care asigura rularea programului cu parametrii ceruți fără a fi necesar ca acesta să fie rulat din linie de comandă, și Working directory – calea spre fișierele sursă.

Astfel, programul se va rula în mod obișnuit folosind butonul Run.

Pentru a configura SparkCotext în prima parte a codului se găsește

```
SparkConf conf = new SparkConf()
        .setAppName("JavaSVMWithSGDExample")
        .setMaster("local");
SparkContext sc = new SparkContext(conf);
```

Ceea ce duce la configurarea unui context Spark folosind obiectul SparkConf care stochează parametrii de configurare pe care aplicația Spark îi va trimite la SparkContext. Unii dintre acești parametrii definesc proprietăți pentru aplicația Spark, iar alții sunt folosind pentru alocarea de resurse in cluster. Cum ar fi numărul, dimensiunea memoriei și nucleele folosite de executorii care rulează pe serverele de lucru (worker nodes).
setAppName( ) - dă aplicației Spark un nume astfel că aceasta poate fi identificată în Spark UI.
*setMaster(local[*]), unde * = ‘ ‘, 2, .. , n* – setează masterul pentru a rula local, dacă local e folosit fără [] atunci Spark rulează pe un singur thread-: nu există paralelism, dacă avem local[i] atunci Spark rulează cu i thread-uri – ideal e să nu se depășească numărul de core-uri ale mașinii.

În momentul în care un SparkConf se transmite la un SparkContext aplicația știe cum să acceseze cluster-ul, ce manager de resurse să folosească și poate cere resurse în cluster.

SparkSession a apărut o dată cu Apache 2.0 și este echivalentul lui SparkContext, eliminând orice confuzie care a existat în trecut pentru Spark în privința contextului pe care îl avea de folosit (metoda .getOrCreate()).

```
SparkSession spark = SparkSession
        .builder()
        .appName("JavaWordCount")
        .config("spark.master", "local")
        .getOrCreate();
```

Un obiect de tip SparkSession poate fi creat folosind un model de constructor (SparkSession.builder). Acesta va refolosi în mod automat un SparkContext dacă acesta există, sau va crea altul nou. Parametrii setați sunt aceeași cu cei descriși anterior.
Execuția algoritmilor a fost realizată pe două laptopuri și un cluster în Google Cloud. Configurațiile acestora se găsesc în tabelul 11.1.
Analiza și interpretarea rezultatelor

- cel mai bun timp de execuție pentru algoritmul/problema ”Word Count” este cel obținut de mașina 2, urmat de mașina 1 si apoi de google cloud cluster.
- pentru algoritmul/problema ”Page Rank” performanța în timp a mașinilor este după cum urmează. Cel mai bun timp este a mașinii ”Local 2”, urmat de Cluster-ul Google Cloud și în final mașina ”Local 1”.
- creșterea numărului de thread-uri/core-uri pentru rularea problemelor descrise a avut ca rezultat o micșorare a timpului de execuție. Acest lucru este de așteptat pentru că mai mulți workeri vor lucra în paralel la aceeași problemă realizând mai repede sarcina.
- pentru a îndeplini bine o sarcină, workerii trebuie să comunice între ei, ceea ce costă timp. Deși, în general, un număr mai mare de workeri înseamnă un timp de execuție mai mic, acest fapt are o limită și o să se constante o creștere a timpului de execuție în loc ca acesta să scadă în continuare, lucru vizibil și în Figura 15. Acest lucru se datorează faptului că procesul de comunicare va necesita tot mai mult timp odată cu creșterea numărul entităților care comunică (workeri). Deci când se adresează probleme de paralelizare în general cât și în Spark trebuie să se țină cont de capacitățile hardware ale mașinii, respectiv numărul de coruri/threaduri.
- aspectul precizat mai sus se aplică și în cazul Cluster-ului în Google Cloud. Deși puterea mașinilor e asemănătoare, comunicarea în interiorul cluster-ului dintre master și workeri generează overhead, tradus întru-un timp de execuție mai mare decât timpul de execuție pe o mașină locală, vizibil și în Figura 11.13.

11.4. Întrebări teoretice

11.4.1. Descrieți modelul de execuție Spark
11.4.2. Ce este un RDD, cum structureaza conținutul d edate și ce rol joacă în arhitectura Spark
11.4.3. Explicați și exemplificați ce diferențe există între transformări și acțiuni?

11.5. Probleme propuse

Se vor implementa , testa și analiza algoritmii prezentați în context de execuție local și cloud.Se vor explica timpii de execuție rezultați, factorii ce determină valorile acestora și modalități posibile de îmbunătățire a lor. Exemplele le puteți descărca de la adresa
https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/11_Spark

11.6. Referințe bibliografice

1. *Surse de date - Data Set*
   movies.txt: https://snap.stanford.edu/data/web-Movies.html
   web-Google.txt: https://snap.stanford.edu/data/web-Google.html
   urls_dataset.txt: http://archive.ics.uci.edu/ml/datasets/URL+Reputation

2. *Documentație Spark* : http://spark.apache.org/docs/latest/
3. *Spark API* http://spark.apache.org/docs/latest/index.html
4. *Tutorial* :https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html
5. *Ghid de programare a aplicațiilor* :https://spark.apache.org/docs/latest/graphx-programming-guide.html
6. Ref card pentru dezvoltatori : https://dzone.com/refcardz/apache-spark?chapter=1

Local 1		
CPU	Intel i5-4210U	1,7 GHz up to 2,7 Ghz, 2 nuclee, 4 threaduri
RAM	8 GB	1333 MHz
Storage	SSD Samsung Evo	citire: 540 Mb/s, scriere: 520 Mb/s

Local 2		
CPU	Intel i7-7700HQ	2,8 GHz up to 3,8 Ghz, 4 nuclee, 8 threaduri
RAM	24 GB	2133 MHz
Storage	SSD WesternDigital Green	citire: 540 Mb/s, scriere: 470 Mb/s

Google Cloud	
Master	4 vCPU, 15.0 GB memory
Workers	2 vCPU, 13.0 GB memory

Timpii de execuție cât și fișierele pe care s-au testat algoritmii “Word Count”, “Page Rank” și ”SVM” :

Tabelu1 11.1. Execuția algoritmilor în diferite contexte de infrastruc

WordCount

File Name	File Size	Total words	Time to complete - Local 1 (average of 5 iterations)				Time to complete - Local 2 (average of 5 iterations)				Time to complete - Google Cloud
			Number of Proccesor Cores				Number of Proccesor Cores				1 master 2 workers
			1	2	4	8	1	2	4	8	
oscar_tanti_roz.txt	82.6 KB	14480	1.383 s	1.333 s	1.405 s	1.524 s	0.657 s	0.659 s	0.676 s	0.673	1.926 s
movies.txt	8.69 GB	6245867	760.047 s	506.486 s	444.606 s	490.758	580.078 s	311.185 s	201.884 s	151.811 s	1074.248 s

PageRank

FileName	File Size	Iterations in page rank	Nodes	Links	Time to complete - Local 1 (average of 5 iterations)				Time to complete - Local 2 (average of 5 iterations)				Time to complete - Google Cloud
					Number of Proccesor Cores				Number of Proccesor Cores				1 master 2 workers
					1	2	4	8	1	2	4	8	
pagerank_data.txt	1 KB	10	4	6	1.756 s	1.803 s	1.749 s	1.839 s	0.954 s	0.871 s	1.073 s	0.893 s	2.001 s
web-Google.txt	71.8 MB	10	875713	5105039	237.796 s	217.310 s	197.698 s	195.401 s	94.624	58.02 s	43.858 s	38.197 s	150.226 s

SVM With SGD

FileName	File Size	Training Iterations	Nr. Instances	Nr. Attributes	Time to complete - Local 1 (average of 5 iterations)				Time to complete - Local 2 (average of 5 iterations)				Time to complete - Google Cloud
					Number of Proccesor Cores				Number of Proccesor Cores				1 master 2 workers
					1	2	4	8	1	2	4	8	
sample_libsvm_data.txt	102 KB	10	-	-	3.419 s	3.860 s	3.899 s	3.798 s	1.354 s	1.57 s	1.481 s	1.435 s	2.205 s
urls_dataset.txt	2.05 GB	10	2396130	3231961	1600.974	1196.818 s	949.078 s	957.934 s	1055.256	573.285 s	320.492 s	261.07 s	630.21 s

Lista Tabele

Lista Figuri