

COSMINA IVAN



**TEHNOLOGII ȘI APLICAȚII
ÎN
CALCULUL PARALEL ȘI DISTRIBUIT**

**UTPRESS
CLUJ-NAPOCA, 2019
ISBN 978-606-737-390-5**

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

Copyright © 2019 Editura U.T.PRESS

Reproducerea integrală sau parțială a textului sau ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

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 diversilor 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 curricula 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/RabbitMQ), 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 română din anul IV al secției de Calculatoare, iar pe această cale doresc să le aduc mulțumiri sprijinului oferit.

Cluj- Napoca, 30 iunie 2019

Cuprins

| | |
|--|-----|
| 1.Programare multithread | 1 |
| 2.Java Concurrency- mecanisme pentru paralelism și concurență | 14 |
| 3.OpenMP - programarea paralelă a sistemelor cu memorie partajată | 27 |
| 4.MPI - programarea paralelă a mașinilor cu memorie distribuită | 40 |
| 5.CUDA - programare masiv paralelă folosind procesoare grafice | 53 |
| 6.Programare distribuită cu socketuri | 69 |
| 7.Programare distribuită prin invocarea metodelor obiectelor remote | 79 |
| 8.Sincronizare și coordonare în sisteme distribuite folosind algoritmi distribuiți | 94 |
| 9.JMS și Jgroups - tehnologii pentru comunicație de grup | 106 |
| 10. Servicii de mesagerie distribuită | 122 |
| 11.Spark –procesare paralelă în context distribuit | 131 |

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 execuție (engl. threaduri) fac trecerea de la programarea secvențială la programarea concurentă/paralelă. Java integrează mecanismele programării concurente sub forma unor clase și interfețe specifice, oferind astfel suportul necesar programării aplicațiilor moderne ce necesită threaduri multiple. Se oferă astfel suport pentru creșterea performanțelor aplicațiilor prin implementarea *paralelismului real* pentru sisteme *multiprocesor*, respectiv a *paralelismului prin întrețeserea threadurilor* pentru sisteme monoprocesor.

Suportul multithread este implementat de mașina virtuală Java (JVM), iar manifestarea threadurilor este direct influențată 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 același timp concurent, alocând periodic *cuante* din timpul de lucru al CPU fiecărui proces. Am reamintit acest lucru deoarece noțiunea de fir de execuție nu are sens decât în cadrul unui sistem de operare multitasking. Un fir de execuție este similar unui proces secvențial în sensul că are un început, o secvență de execuție și un sfârșit. Un program își poate defini însă nu doar un fir de execuție ci oricâte, ceea ce înseamnă că în cadrul unui proces se pot executa simultan mai multe fire de execuție, permițând astfel execuția concurentă a sarcinilor independente ale programului.

Execuția simultană a firelor de execuție în cadrul unui proces este similară cu execuția concurentă a proceselor: sistemul de operare va alocă ciclic cuante din timpul procesorului fiecărui fir de execuție 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 execuție nu este copiat decât codul procesului părinte; toate firele de execuție au deci *acces la aceleași date*, datele procesului original. Deasemenea, comutarea de context și comunicarea între threaduri este mult mai simplă și mai rapidă.

Proiectarea și implementarea aplicațiilor folosind threaduri oferă o serie de avantaje, astfel:

- simplificarea programării unor aplicații mai complexe și o mai bună structurare a codului
- eficiență în utilizarea resurselor sistemului (modelul multithread poate înlocui soluțiile clasice de comunicare între procese de tip IPC pentru anumite probleme)
- ascunderea latenței acceselor la memorie, I-O sau comunicației și astfel câștig de performanță prin suprapunere comunicație-procesare
- îmbunătățirea interacțiunii la nivel de aplicație prin proiectarea unor interfețe 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 și echilibrarea încărcării de execuție 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 și astfel un proces master creează alte procese explicit pentru a putea separa logic funcționalitățile aplicației. Dacă procesele pot fi considerate construcții ce determină arhitectura aplicației, threadurile sunt construcții specifice limbajului.

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

- *wait()* - pune obiectul în așteptare până la apariția unui eveniment (notificare) cu/fără indicarea duratei maxime de așteptare
- *notify()* - permite anunțarea altor obiecte de apariția unui eveniment
- *notifyAll()* - implementează mod broadcast notificarea mai multor obiecte la apariția unor evenimente
- *finalize()* - poate fi suprascrisă în clase și definește diversele acțiuni ce vor fi executate de “colectorul de deșuri” JVM înaintea distrugerii obiectului

Clasa Thread. Mașina virtuală Java permite definirea mai multor threaduri concurente, orice thread este un obiect Java ce posedă metode și poate fi transferat ca parametru, plasat într-un tablou, etc. Mașina 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 și anume: *identificatorul de thread*, *numele*, *politica de planificare*, *prioritatea*, cine este responsabil cu controlul execuției threadului (procesul utilizator sau nucleul sistemului de operare). Un thread există în cadrul unui proces, el este compus din context, structura utilizator conținând copii ale valorilor regiștrilor generali, stiva, *zona de date private* și *setul de instrucțiuni*. Un thread este o entitate planificabilă, ea poate fi întreruptă preemptiv și poate utiliza în funcție de numărul de procesoare din sistem concurența 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 același 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 execuția cu *metoda run()* similar începerii execuției unui program cu metoda *main()*
- orice thread posedă o *anumită prioritate* (valoare întreagă între Thread.MIN_PRIORITY și 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. Mașina virtuală mapează prioritățile threadurilor nivelurilor de prioritate ale platformei care poate dispune de mai puține niveluri de prioritate (sub Windows, anumite niveluri JVM vor fi mapate acelorași 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 și în absența acestuia.

Principalele câmpuri și metode ale clasei Thread sunt:

- *void start()* - lansează în execuție noul thread, moment în care execuția programului este controlată de cel puțin două threaduri: threadul curent ce execută metoda *start* și noul thread ale cărui instrucțiuni sunt definite în metoda *run ()*.

- *void run()* – definește corpul threadului nou creat, întreaga activitate a threadului va fi descrisă prin suprascrierea acestei metode
- *static void sleep ()* – pune în așteptare threadul curent pentru un anumit interval de timp (*msecs*)
- *void join ()* - se așteaptă ca obiectul thread ce apelează această metodă să se termine
- *suspend()* - suspendare temporară a threadului (*resume()* este metoda duală ce relansează un thread suspendat (implementările JDK ulterioare versiunii 1.2 au renunțat 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()* și *interrupted()* permit testarea stării de întrerupere a threadului apelant. Metoda *interrupted()* modifică starea threadului curent (la un apel secund al ei starea threadului revine la cea inițială).
- *static boolean interrupted()* - metodă statică, testează dacă threadul curent a fost întrerupt, resetează starea *interrupted* a threadului curent
- *boolean isInterrupted ()* - testează dacă un thread a fost întrerupt fără a modifica starea threadului
- *static Thread current Thread()* - returnează obiectul reprezentând threadul curent în execuție
- *boolean isAlive()* - permite identificarea stării obiectului thread, astfel metoda returnează *true* dacă threadul a fost pornit și nu a murit încă, respectiv *false* dacă threadul nu a fost pornit sau a murit, fără a putea diferenția 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* conține un ciclu infinit, astfel încât acesta nu se va termina la terminarea threadului părinte.
- *getPriority()* - returnează prioritatea threadului curent
- *setPriority(newPriority)* - permite atribuirea pentru threadul curent a unei priorități dintr-un interval.

Metodele *stop()*, *suspend()* și *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 execuție corectă). O clasă aparte de thread-uri sunt cele Daemon care sunt thread-uri de serviciu (aflate în serviciul altor fire de execuție). Când se pornește mașina virtuală Java, există un singur fir de execuție care nu este de tip Daemon și 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 și invers.

1.2.2. Controlul și ciclul de viață al unui thread

Fiecare fir de execuție are propriul său ciclu de viață: este creat, devine activ prin lansarea sa în execuție și la un moment dat, se termină. În continuare vom vedea mai îndeaproape stările în care se poate găsi un fir de execuție. Diagrama din Figura 1.1. ilustrează aceste stări precum și metodele care provoacă tranziția dintr-o stare în alta. Așadar, un fir de execuție 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 execuție se găsește în această stare imediat după crearea sa, cu alte cuvinte după instanțierea unui obiect din clasa *Thread* sau dintr-o subclasă a sa.

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

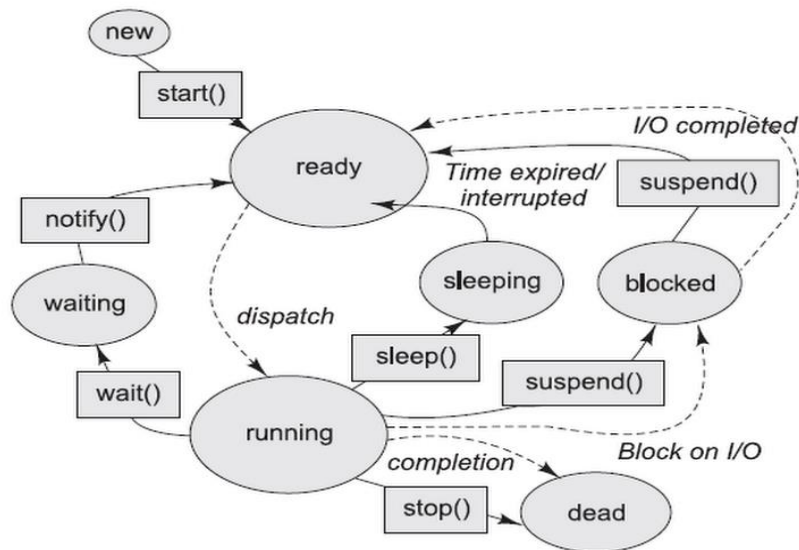



Figura 1.1. Stările unui thread Java [4]

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

Stări de tip "Runnable" (ready/running)

După apelul metodei `start` un fir de execuție va trece în starea "Ready", adică se găsește în execuție.

```
counterThread.start();
//counterThread se găsește în starea ready
```

Metoda `start` realizează următoarele operațiuni necesare rulării firului de execuție:

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

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

Un fir de execuție ajunge în această stare în una din următoarele situații:

- este "adormit" prin apelul metodei `sleep`.
- a apelat metoda `wait`, așteptând ca o anumită condiție să fie satisfăcută
- este blocat într-o operație de intrare/ieșire

Metoda `sleep` este o metodă statică a clasei `Thread` care provoacă o pauză în timpul rulării firului curent aflat în execuție, 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 excepții 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, își continuă execuția. Pentru fiecare tip de intrare în starea "Not Runnable", există o secvență specifică de ieșire din starea respectivă, care readuce firul de execuție în stări Runnable. Acestea sunt:

- dacă un fir de execuție a fost "adormit", atunci el devine Runnable doar după scurgerea intervalului de timp specificat de instrucțiunea sleep.
- dacă un fir de execuție așteaptă o anumită condiție, atunci un alt obiect trebuie să îl informeze dacă acea condiție este îndeplinită sau nu; acest lucru se realizează prin instrucțiunile *notify* sau *notifyAll*
- dacă un fir de execuție este blocat într-o operațiune de intrare/ieșire atunci el redevine Running atunci când acea operațiune s-a terminat.

Starea "Dead"

Este starea în care ajunge un fir de execuție la terminarea sa. Un fir de execuție 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 forțat un fir de execuție, însă ea a fost eliminată din motive de securitate.

1.2.3.Operații cu threaduri Java

a. Crearea și pornirea unui thread

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

Pentru a decide care din cele două metode este necesară unui anumit context de programare este utilă următoarea observație: dacă clasa este derivată din altă clasă este de preferat să implementeze interfața Runnable (datorită lipsei mecanismului de moștenire multiplă în limbajul Java), iar în caz contrar, se utilizează extinderea clasei Thread, ambele soluții 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() moștenită din clasa Thread
3. se instanțiază un obiect thread folosind new
4. se pornește thread-ul instanțiat, prin apelul metodei start() moștenită din clasa Thread. Apelul acestei metode face ca mașina 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 interfața java.lang.Runnable.

Pentru această metodă de implementare, etapele specifice sunt:

1. se creează o clasă care implementează interfața Runnable
2. se implementează metoda run() din interfață
3. se instanțiază 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 interfața)
5. se pornește 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 dorește a fi implementată moștenește o altă clasă (Java nu permite moștenirea multiplă). Interfața Runnable descrie o singură metodă run().

Metoda start() creează resursele sistem necesare execuției threadului și 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 așteaptă 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 ciclului eviață 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 situația în care apare unul din următoarele evenimente:

- este invocată metoda *sleep()*, astfel deși procesorul este disponibil threadul este pus în așteptare un anumit interval de timp, dacă threadul revine în starea Runnable și procesorul redevine disponibil threadul își va reîncepe execuția.
- threadul apelează metoda *wait()* pentru a aștepta satisfacerea unei anumite condiții, iar pentru a deveni Runnable, un alt thread trebuie să notifice threadul în așteptare despre o modificare a condiției apelând *notify ()* sau *notifyAll()*.
- threadul e blocat la o operație I/O și devine Runnable la terminarea operației.

c. Terminarea unui thread

JVM va executa un thread cât și threadurile create de acesta până la întâlnirea uneia din situațiile: 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 recepția unei întreruperi *interrupt()* din partea altui thread. După cum am văzut, un fir de execuție nu poate fi terminat forțat de către program ci trebuie să-și programeze singur terminarea sa. Acest lucru poate fi realizat în două modalități :

1. **prin scrierea unor metode run care să-și termine execuția** în mod natural; la terminarea metodei run se va termina automat și firul de execuție, 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 și printr-o variabilă care să oprească această buclă atunci când dorim ca firul de execuție să se termine. Uzual, aceasta este o variabilă membră a clasei care descrie firul de execuție care fie este publică, fie este asociată cu o metoda care îi schimbă valoarea.

Metoda *run()* nu se termină natural, ea rulează la infinit așteptând să fie terminată forțat folosind de exemplu metoda *stop*. Această metodă este însă "învechită" (deprecated) iar la compilarea programului vom obține 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 execuție a fost pornit și 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 diferențiere.

Nu este necesară distrugerea explicită a unui fir de execuție, sistemul se ocupa de acest lucru. El poate fi forțat să dezaloc resursele alocate unui thread prin atribuirea cu null a variabilei care referea instanța firului de execuție: *myThread = null* .

Așteptarea terminării unui thread. Dacă este necesară așteptarea terminării unui thread se poate utiliza metoda *join()* a obiectului thread după care se așteaptă, funcție ce are același efect ca și combinația *sleep()* și *isAlive()*. Apelul *this.join()* referă o așteptare perpetuă, astfel threadul curent nu se va termina. Funcția *join()* pentru un thread ce nu și/a început execuția sau deja s-a terminat va returna fără să aștepte.

d. Întreruperea unui thread

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

Preemptiv vs. non-preemptiv .Sistemul runtime Java nu va întrerupe threadul curent pentru un alt thread de aceeași prioritate, este posibil însă ca sistemul de operare să implementeze propria politică de preemptie (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 așteptare la o condiție, prin apelul metodelor *sleep* sau invocarea unor operații de intrare-ieșire, este necesară invocarea metodei *yield()*. Prin această metodă, threadul renunță de bună voie la serviciile procesorului și mașina virtuală Java este anunțată că poate preda controlul unui alt thread pe care planificatorul decide să îl treacă în starea ready. În orice moment este posibilă verificarea execuției unui thread folosind funcția *isAlive()*.

e. Planificarea threadurilor

Execuția într-o anumită ordine a mai multor fire de execuție pe un singur procesor se numește *planificare* (engl. *scheduling*). Sistemul Java de execuție a programelor implementează un algoritm simplu, determinist de planificare, cunoscut sub numele de *planificare cu prioritati fixate*. Fiecare fir de execuție Java primește la crearea sa o anumită prioritate. O prioritate este de fapt un număr întreg cu valori cuprinse între `MIN_PRIORITY` și `MAX_PRIORITY`. Implicit prioritatea unui fir de execuție 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 priorității unui fir de execuție se realizează cu metoda `setPriority()` a clasei `Thread`. Planificatorul Java lucrează în modul următor: dacă la un moment dat sunt mai multe fire de execuție 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 execuție cu prioritate maximă se termină sau este suspendat din diverse motive va fi ales un fir de execuție cu o prioritate mai mică. În cazul în care toate firele au aceeași 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 execuție, iar celălalt trecut în așteptare.

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

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

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

f. Grupuri de threaduri

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

Fiecare fir de execuție Java este membru al unui grup, indiferent dacă specificăm explicit acest lucru. Afilierea unui fir de execuție la un anumit grup se realizează la crearea sa și devine permanentă, în sensul că nu vom putea muta un fir de execuție dintr-un grup în altul, după ce acesta a fost creat. În cazul în care creăm un fir de execuție fără a specifica în constructor din ce grup face parte, el va fi plasat automat în același grup cu firul de execuție 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 execuție create direct din program și care nu au fost atașate explicit altui grup. Cu alte cuvinte, putem să ignorăm complet plasarea firelor de execuție în grupuri și să lăsam sistemul să se ocupe cu aceasta.

Exista situații când programul creează multe fire de execuție, iar gruparea lor poate simplifica substanțial manevrarea lor. Un grup se compune dintr-o mulțime de threaduri și poate conține alte grupuri, formând un arbore, un thread poate accesa doar informația referitoare la propriul grup fără a avea acces la alte grupuri sau la grupul părinte. Orice thread aparține unui grup specificat ca un parametru constructor sau moștenit. Threadurile unui grup pot fi enumerate, pentru un grup poate fi setată prioritatea maximă respectiv grupul de threaduri poate fi întrerupt din execuție. Grupurile de threaduri sunt folosite deoarece permit manipularea mai multor threaduri la un singur apel de funcție și oferă bazele mecanismului de securitate folosit pentru lucrul cu threaduri. În mod implicit toate threadurile create în program aparțin unui grup definit de mașina 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 semnificației pentru grupuri. Un constructor de thread, în forma cea mai generală atașează noul thread unui grup de threaduri, dacă argumentul lipsește atunci este atașat grupului curent. Grupul unui thread este stabilit la crearea acestuia și nu mai poate fi modificat pe parcursul execuției sale, la terminare threadul este automat eliminat din grup Pentru a afla cărui grup aparține un anumit fir de execuție putem folosi metoda `getThreadGroup` a clasei Thread. Un grup poate avea ca părinte un alt grup, ceea ce înseamnă că firele de execuție 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 execuție 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 octeți
 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 ieșire 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 operații de scriere/citire folosirea lor se va face în cadrul unor fire de execuție. Funcționarea obiectelor care instanțiază *PipedWriter* și *PipedReader* este asemănătoare cu a canalelor UNIX (pipes). Fiecare capăt al unui canal este utilizat dintr-un fir de execuție separat. La un capăt al pipeline-ului se scriu caractere, la celalalt se citesc. La citire, dacă nu sunt date disponibile firul de execuție se va bloca. Se observă că acesta este un comportament tipic producător-consumator, firele de execuție 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 și citirea pe/de pe canale se realizează prin metodele uzuale `read` și `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 secțiunea precedent s-a prezentat cum pot fi create fire de execuție *independente și asincrone*, care nu depind în nici un fel de execuția sau de rezultatele altor fire de execuție. Există însă numeroase situații când fire de execuție separate, dar care rulează concurrent, trebuie să comunice între ele pentru a accesa diferite resurse comune sau pentru a-și 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 concurrent este necesară chiar dacă sistemul este uniprocessor. Blocul de cod ce accesează același obiect din două threaduri separate este numit *secțiune critică*. Pentru a permite unui singur thread accesul la secțiunea 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 potențial monitor având asociată o coadă de așteptare și mecanisme de semnalizare.

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

Referitor la utilizarea metodelor-secțiunilor de program sincronizate cât și pentru a evita condițiile de tip “cursă” în accesul la date, asigurând astfel integritatea acestora se impun următoarele precizări:

- *secțiune critică* poate reprezenta o *metodă* sau un *bloc* de cod și 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 și eliberarea lacătului este realizată în mod *automat și atomic* de către sistemul runtime la intrarea, respectiv ieșirea din blocul de cod declarat *sincronizat*.
- modificările efectuate într-o metodă sincronizată devin *vizibile* celorlalte metode sincronizate referitor la același 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 până la eliberarea lacătului, acestea vor fi blocate.
- atunci când threadul ce deține lacătul va reveni din metoda sincronizată lacătul obiectului va fi în mod automat eliberat
- unul din threadurile ce așteaptă acapararea lacătului îl va prelua și va apela metoda sincronizată.
- citirile și scrierile unor câmpuri volatile sunt atomice și 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 execuției metodei
- mecanismul de sincronizare a blocurilor de cod, permite specificarea explicită a lacătului unui anumit obiect ce trebuie acaparat de un thread înaintea execuției 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)
- creșterea gradului de paralelism poate fi implementată prin declararea unui întreg bloc de cod ca *synchronized*.

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

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

Anunțul (notificarea) efectuat cu metoda *notify()* este adresat tuturor threadurilor aflate în starea Waiting, dar unul singur își va relua activitatea, iar pentru un anunț efectuat cu *notifyAll()* toate threadurile în așteptare își 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 așteptare. Dacă un alt thread apelează metoda *interrupt()* a threadului în așteptare, execuția acestuia este reluată cu secvența *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 condiționale au asociate variabile tip mutex pe care le blochează-deblochează funcție de faza așteptării. Astfel în Java înaintea intrării în așteptare prin *wait()*, threadul curent eliberează obiectul de sincronizare blocat și îl obține din nou când threadul este anunțat că a fost realizată condiția așteptată. 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 aștepte, se va reveni fără eroare
- metoda *wait()* eliberează obiectul de sincronizare la intrarea în așteptare și îl reobține la recepționarea notificării
- dacă există mai multe threaduri ce așteaptă o anumită condiție nu se poate ști care thread va primi notificarea, mașina virtuală Java și mecanismul de planificare a threadurilor determină ordinea precisă
- dacă se dorește apelul funcțiilor *wait()*-*notify()* din metode statice este necesară declararea în cadrul clasei a unui obiect de sincronizare static pentru care vor fi apelate aceste funcții
- diferența între metodele *wait()* și *sleep()* constă în aceea că metoda *sleep* nu necesită apelul obligatoriu dintr-un bloc sincronizat și 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 deține încă lacătul. Funcția va transfera toate threadurile aflate în așteptare din coada de așteptare în coada de blocare așteptând momentul în care pot continua.

Evitarea metodelor stop() și suspend(). Metodele *stop*, *resume* și *suspend()* lasă obiectele în stări inconsistente. Un thread suspendat forțat poate deține infinit un lacăt, iar dacă va apela ulterior *resume()* va genera blocaj datorat aceluiași lacăt. Threadul trebuie să verifice dacă a fost întrerupt. Versiuni sigure pentru operațiile *suspend()* și *resume()* pot fi implementate bazat pe operațiile *wait()* și *notifyAll()*, astfel threadul va verifica cererea de suspendare și va controla momentul în care își poate întrerupe în siguranță activitățile.

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

Următorul set de reguli permit evitarea blocajelor în secvențele de cod ce necesită concurență:

- evitarea apelurilor blocante (operații de intrare-ieșire) în cadrul unui bloc sincronizat
- evitarea apelurilor imbricate sincronizate
- structurarea liniară a fluxului de control prin aplicarea unor strategii și/sau euristici între care reordonarea resurselor, creșterea granularității (obținerea 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 cerințelor de atomicitate totală (ex. propagarea modificărilor și 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, prioritzare 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 prioritzarea 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 soluție poate fi considerată utilizarea unei cozi intermediare, se cunoaște 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 facilitățile de sincronizare specifice limbajului Java problema accesului concurrent a scriitorilor și cititorilor la un fișier. Realizați o soluție ce oferă acces *concurrent* pentru citire și exclusiv pentru scriere.

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

1.5.4. Scrieți un program Java care generează două fire de execuție pentru parcurgerea unui String de la cele două capete. Folosiți doi pointeri a căror valoare se incrementează, respectiv se decrementează într-o funcție 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.concurrent* 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`. Îmbunătățirile 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` și `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 thread-uri (*thread pool*). Astfel, *work queue* este o coadă de task-uri 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 managementul 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.

Interfața 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 folosite 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 permisiile ce pot fi cerute și eliberate. Acesta este folosit pentru a restricționa numărul de thread-uri ce pot avea simultan acces concurrent 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 achiziționeze 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* și *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 aceeași resursă.

Conditions. La fel cum interfața *Lock* este generalizarea pentru *synchronized*, interfața *Condition* este o generalizare a metodelor *wait()* și *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*. Așadar, deși sunt transparente pentru majoritatea utilizatorilor, această categorie de clase este cea care aduce principalele îmbunătățiri 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 aplicațiile moderne.

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

- *java.util.concurrent.atomic* - ce oferă o mai mare flexibilitate în utilizarea de lacăte și condiții, în detrimentul sintaxei clasice
- *java.util.concurrent.locks* - clasele din acest pachet extind noțiunea de valoare volatilă, câmpuri și șiruri de elemente cărora, de asemenea, le oferă operații atomice

a)Executori – *java.util.concurrent.Executor*

Atunci când este necesar să fie rulate mai multe sarcini complexe, în paralel și să se aștepte 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 soluționa această problemă, Java introduce Executor, o interfață ce permite crearea seturilor de thread-uri, susține sincronizarea și execuția lor.

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

- implementarea interfeței
- crearea unui obiect ce implementează interfața 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(); // execuția va avea loc în threadul apelant
        new Thread(obj).start(); // execuția va avea loc într-un thread distinct }}

```

Dacă este nevoie de generarea și 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 instanță a clasei ExecutorService. Acesta poate fi de mai multe tipuri :

- *Single Thread Executor* – un set care conține un singur thread; codul se va executa secvențial
- *Fixed Thread Pool* – un set care conține un număr fix de thread-uri; dacă un thread nu este disponibil pentru un task, acesta se pune într-o coadă și așteaptă finalizarea unui alt task
- *Cached Thread Pool* – un set care creează atâtea thread-uri câte 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 conține 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 *newThread* (Runnable r) care se suprascrive în clasele care o implementează, pentru crearea unor thread-uri personalizate. Un exemplu de utilizare al acestei interfețe îl găsiți 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

Coadă (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*, *ArrayBlockingQueue*, *BlockingQueue*, *ConcurrentLinkedQueue*, *DelayQueue*, etc.

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 și șterge elementul din vârful cozii; dacă nu mai există elemente în coadă returnează null
- *take()* throws *InterruptedException* - returnează și șterge elementul din vârful cozii; dacă nu mai există elemente în coadă blochează thread-ul care a lansat operația
- *void clear()* - șterge 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 aceleași resurse simultan. Din acest motiv au fost introduse mai multe metode de sincronizare prezentate în cele ce urmează.

d) Excluderea reciprocă

Modificatorul *synchronized* atașat 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 și 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 definește 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 semnaliza 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 văzută 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)*, *CyclicBarrier (int num, Runnable barrierAction)* 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    recruiting,    "    +
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 și 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 și 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* și 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 și 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 și 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, separați prin virgulă și cuprinși eventual între paranteze rotunde,
- săgeata direcțională ->
- un corp ce constă dintr-o expresie sau un bloc de instrucțiuni.

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 și 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 și 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 / concurrentă 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 execuție*, *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 Analyzer), 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 să 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 Concurrency 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_concurrent.

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 deservește un număr nelimitat de clienți folosind FW Executor cu parametrizare adecvată. Se va utiliza analizorul de performanță Netbeans Profiler și se va urmări modificarea timpului de execuție ca urmare a modificării parametrilor din frameworkul Executor.

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

2.6. Miniproiect

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

Modelul boss-worker este cel mai potrivit pentru aplicațiile de tipul producător - consumator. În acest model, un thread este desemnat ca boss, toate celelalte fiind desemnate ca workers. Threadul boss obține sau produce sarcini și le plasează într-o coadă. Threadurile worker preiau sarcinile din coadă și le procesează. Exemple de aplicații în care acest model este recomandat: aplicații care primesc input din surse externe (ex. servere, interfețe cu utilizatorul), algoritmi de căutare, algoritmi de tipul divide et impera, procesarea buclor în paralel. Deși în general există un singur thread boss, nimic nu împiedică o aplicație să creeze mai multe threaduri boss. De asemenea, oricare dintre threadurile worker poate acționa 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 utilizează în general un "**thread-pool**". Astfel thread-ul (threadurile) boss produce sarcini într-o coadă pe care apoi thread-urile worker le procesează. Un threadpool adresează două probleme:

1. îmbunătățirea performanțelor în cazul în care un număr mare de taskuri sunt executate prin reutilizarea threadurilor worker;
2. limitarea și managementul resurselor utilizate pentru execuția unei colecții de sarcini (taskuri).

Modelul boss-worker în abordarea clasică, anterior JDK 1.5 se implementează utilizând: două variabile condiționale ,un mutex ,o coadă ,un număr de contoare .Mutexul protejează coada, variabilele condiționale și contoarele. O variabilă condițională este folosită de threadurile worker în așteptare atunci când coada este goală. Cealaltă variabilă condițională este utilizată de threadul boss atunci când coada este plină. Un minimum de 3 contoare este necesar pentru a cunoaște în permanență: C1-numărul de sarcini existente în coadă, C2-numărul de threaduri worker în așteptare C3-numărul de threaduri boss în așteptare

Pentru a plasa noi sarcini în coadă, threadul boss trebuie întâi să preia controlul asupra mutexului. În cazul în care coada nu este plină, threadul boss adaugă sarcini în coadă și semnalează acest lucru eventualelor threaduri worker care așteptau. În cazul în care coada este plină, threadul boss așteaptă la o variabilă condițională până când se face loc în coadă pentru noi sarcini. Odată ce sarcinile au fost depuse în coadă, mutexul este eliberat și threadul boss se reîntoarce la munca sa (adică la crearea sau obținerea 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 și acest lucru este semnalat eventualelor threaduri boss care așteptau. Dacă nu există sarcini în coada (coada este vidă), threadul worker așteaptă la o variabilă condițională până când apar sarcini în coadă. Odată ce sarcina a fost preluată din coada, mutexul este eliberat și threadul worker începe să proceseze sarcina sa. Sincronizarea este necesară numai atunci când au loc operații 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. În timp ce threadul boss creează sarcini, nici unul dintre threadurile worker nu poate accesa această structură sau datele asociate ei. Odată ce un thread worker și-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 predefinită, sunt necesare două variabile condiționale (una pentru threadurile worker, cealaltă pentru threadurile boss). Există situații în care lungimea cozii poate fi infinită. În 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 situații este cel menționat și anume că threadul boss se poate bloca, fapt ce poate avea diferite impacte asupra utilizatorului sau timpului de răspuns al aplicației.

În modelul boss-worker, fiecare dintre threadurile worker se află într-o buclă continuă în care obțin și procesează 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.

Există două tehnici de realizare a acestui lucru:

1. să 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 setează 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 să fie vidă. Dacă 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 împărți sarcinile astfel încât atunci când un thread se blochează, un altul poate continua în locul lui. Acest caz este cel mai des întâlnit atunci când modelul boss-worker este folosit pentru servere și interfețe cu utilizatorul, unde cantitatea de sarcini este controlată de evenimente externe. Threadul boss acceptă taskuri și le pasează threadurilor worker, iar planificarea balansării nu este o problemă pentru aceste aplicații.

Un alt tip de echilibrare este totuși important în modelul boss-worker: echilibrarea cozii. O aplicație eficientă 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.

În 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 în 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 parametri pot fi specificați la instanțierea unui astfel de obiect:

- **corePoolSize** - numărul minim de fire de execuție "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 decât `corePoolSize`, acest parametru specifică timpul maxim în care firele de execuție în exces vor aștepta 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 interfața Runnable și poate fi introdusă în coadă utilizând metoda execute.

Coadă trebuie să implementeze interfața BlockingQueue<E> Implementări existente pentru o coadă pe care puteți să le utilizați:

ArrayBlockingQueue -coadă de dimensiune fixă ce utilizează un vector pentru salvarea sarcinilor
LinkedBlockingQueue -coadă de dimensiune nelimitată implementată printr-o listă simplu înlanțuită (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 execuție worker
- **handler** - handler utilizat pentru a primi notificari în cazul în 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://howtodojava.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 mașinilor cu memorie partajată, oferind paralelism de tip fork-join explicit în care un thread master creează și coordonează multiple threaduri, model parametrizabil prin intermediul unor construcții de tip directive și clauzele asociate acestora , respectiv utilizând variabile de mediu și funcții de bibliotecă pentru crearea de taskuri paralele/concurente cu execuție controlată prin construcțiile 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 ARB www.openmp.org) și a evoluat continuu ajungând în prezent la versiunea 5.0.(2018) , fiind utilizat pentru dezvoltarea aplicațiilor paralele pe platforme diverse de la sisteme desktop până la supercomputere. În cadrul acestui model, threaduri cooperante rezolvă sarcini (taskuri) prin execuție simultană pe procesoare sau core-uri multiple.

Modelul de programare OpenMP se bazează pe:

- *memorie partajata și paralelism bazat pe threaduri* : un proces cu memorie partajată poate consta în fire de execuție multiple. OpenMP se bazează pe existența firelor multiple în paradigma programării cu partajarea memoriei.
- *paralelism explicit*: OpenMP este un model de programare explicit (nu automat), care oferă programatorului deplinul control asupra paralelizării.
- *modelul fork-join* : OpenMP utilizează de un model al execuției paralele alcătuit din ramificații și joncțiuni vezi (vezi Figura 3.1.)

Directivele compiler sunt utilizate ca extensii pentru limbajele secvențiale C++/Fortran oferind construcții pentru crearea de taskuri, partajarea sarcinilor , sincronizarea threadurilor .

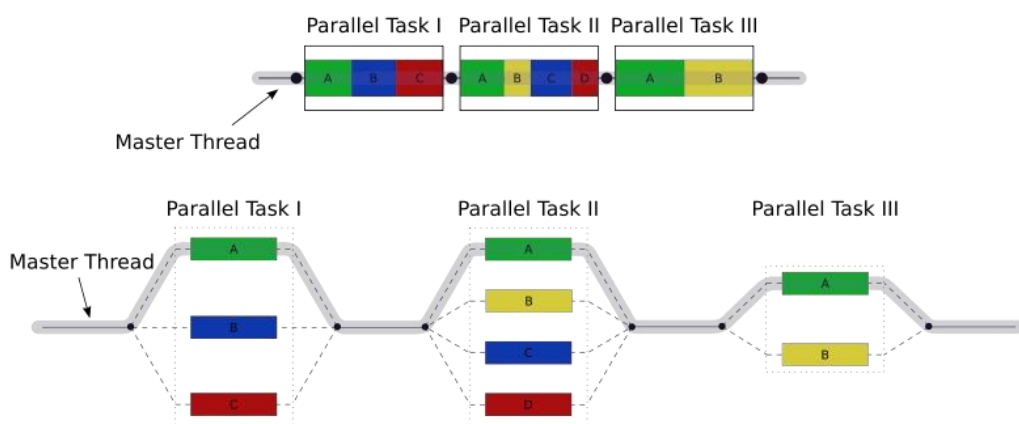


Figura 3.1. Modelul fork-join OpenMP [1]

Setul de directive OpenMP permite specificarea acțiunilor compilator necesare execuției codului program în paralel fără însă a verifica *dependențele, conflictele, blocajele* posibile, programatorul fiind astfel responsabil în totalitate de proiectarea corectă a aplicației. Modelul nu este aplicabil sistemelor cu memorie distribuită, nu este implementat în mod identic de către toți producătorii (de ex. Intel/Microsoft) și 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 așteptat a deveni în viitor un standard ANSI.

3.2.1. Modelul de programare și modelul de memorie

Standardul OpenMP oferă o interfață (API) de *programare independentă de platformă* (există implementări Linux/Windows), integrând o serie de construcții pentru managementul proceselor, distribuția taskurilor, concurență și sincronizare a threadurilor și gestionarea datelor. Modelul de programare oferă deasemenea construcții pentru controlul partajării și consistenței memoriei utilizate de aplicație și suport pentru paralelism imbricat.

Regiunea paralelă este construcția ce definește o secțiune de program sub forma unui *bloc structurat* (marcat prin intermediul acoladelor), *executat în paralel de procesoarele sistemului*. Programul își începe execuția 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.) și se include pe sine în echipă, threaduri ce vor executa regiunea paralelă prin planificarea (alocarea) iterațiilor ciclurilor paralele acestor threaduri. Această operație de tip fork presupune că la finele regiunii paralele există o barieră *implicită* pentru sincronizare și doar threadul master continuă execuția ulterior acesteia. *Numărul de threaduri* poate fi specificat în directivă, stabilit folosind variabile de mediu sau în mod dinamic la runtime prin funcții OpenMP. Dacă un thread modifică un obiect partajat, acest fapt va afecta nu doar propriul context de execuție ci și pe cele ale celorlalte threaduri ale programului care au acces și 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 și 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 funcții runtime și variabile de mediu (vezi Figura 3.2.) și vor fi abordate în continuare.

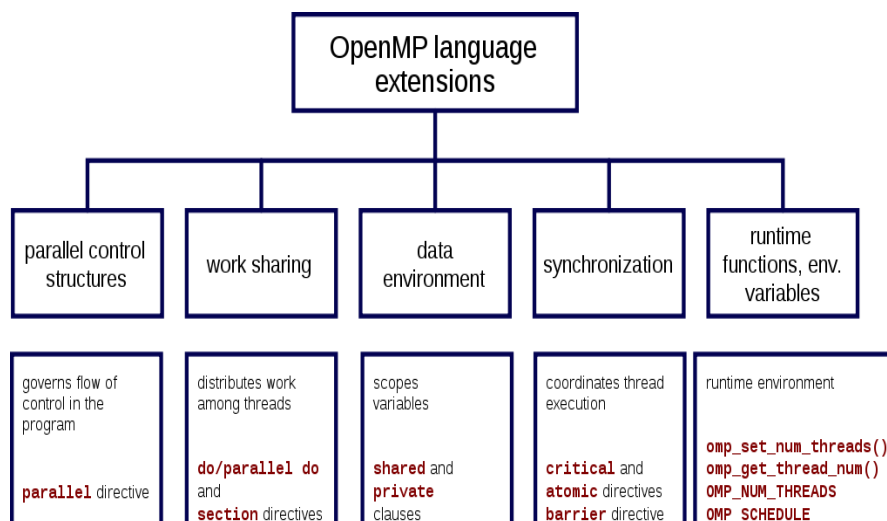


Figura 3.2. Tipuri de construcții OpenMP [1]

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

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

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

Clauze. Specificarea de informații adiționale 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 condiționale, a numărului de threaduri de execuție și a modului de gestionare a datelor, cele mai importante categorii de clauze fiind:

- Specificarea datelor : acestea pot fi date partajate și private. În regiunile paralele, variabilele pot fi *partajate sau private*. Toate thread-urile văd aceeași 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 aparține.
- Specificarea ciclurilor paralele. Ciclurile sunt principala sursă de paralelism, dacă iterațiile unui ciclu sunt *independente* (pot fi executate în orice ordine), atunci se pot partaja iterațiile între thread-uri diferite.
- Specificare unor operații pe date ,de exemplu reducții. Reducția este operația prin care se generează o singură valoare din operații asociative. Permițând 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 condițională. Directivele respectă convențiile 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 declarație 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 și finalul unui bloc structurat care urmează directiva. Domeniul static al unei directive nu acoperă rutine multiple sau fișiere 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 și posibile fișiere de cod.
- *domeniu dinamic:* Domeniul dinamic al unei directive include atât domeniul ei static (lexical) cât și domeniile orfanelor ei

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

OMP_NUM_THREADS -setează numărul de threaduri posibil a fi utilizate în timpul execuției 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 și private. În interiorul unei regiuni paralele variabilele pot fi *partajate* -toate thread-urile văd aceeași copie (modelul implicit), sau *private* - fiecare thread are propria copie asupra căreia operează și al cărei conținut poate 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 observații se impun referitor la declararea tipurilor variabilelor:

- majoritatea variabilelor sunt partajate (este modelul implicit)
- indicii ciclurilor sunt privați
- 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(operator: listă_var)
num_threads(expresie_int.)
```

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

Există o serie de funcții complementare cu ajutorul cărora poate fi controlată execuția 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, funcția returnează 1 dacă este apelată în exteriorul unei regiuni paralele
int omp_get_thread_num(void) - determină numărul threadului curent, ia valori între 0 și 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) - funcție ce returnează limita superioară a numărului de threaduri ce poate fi creat de directiva PARALLEL
int omp_get_num_procs(void) - funcție ce returnează numărul maxim de procesoare ce pot fi utilizate pentru execuția programului
```

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

Inițializarea variabilelor private. Atunci când o variabilă este declarată privată, threadul primește adresa unde va stoca valorile variabilei pe durata regiunii paralele. La încheierea regiunii paralele, memoria este dealocată și aceste variabile nu mai există. Dacă se dorește reținerea valorii acestor variabile anterior/ulterior regiunii paralele se vor folosi clauzele `FIRSTPRIVATE` și

LASTPRIVATE. Variabilele private sunt neinițializate la începutul regiunii paralele. Dacă se dorește inițializarea lor, se poate folosi clauza FIRSTPRIVATE: `firstprivate(list)`.

Clauza de reducere. O reducere produce o singură valoare din operații asociative precum: adunarea, înmulțirea, max, min, ȘI (logic), SAU (logic). Operația 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 șirurilor ca variabile de reducere.

Sintaxa clauzei este: `reduction(op:list)`

Clauza IF. Directiva de regiune paralelă poate fi condițională, specificare utilă situațiilor 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 execuția între membrii echipei de threaduri. Aceste directive sunt plasate în interiorul unei regiuni paralele și indică modul în care este divizat lucrul. Directivele nu lansează noi threaduri, iar la începutul unei asemenea construcții nu se consideră în mod implicit plasarea unei bariere, secvența de construcții de partajare a lucrului și 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 iterațiilor ciclului între thread-uri*. Ciclurile identifică o construcție de partajare a sarcinilor iterativă ce permite specificarea execuției în paralel a iterațiilor asociate ciclului și 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(operator: listă _var)  
ordered  
schedule(tip[, chunk_size])  
nowait
```

Semnificația clauzelor în contextul acestei directive este următoarea: *firstprivate* specifică copii locale ale variabilei private threadului, iar *lastprivate* permite specificarea actualizării variabilei la ultima iterație a ciclului for. Directiva plasează restricții 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: `<`, `<=`, `>`, `>=`

și `incr-expr` este `var = var +/- incr` sau echivalenți semantici precum `var++`, ce nu poate fi modificată în corpul ciclului. Forma canonică permite ca numărul de iterații să fie determinat la intrarea ciclului.

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

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

Fără clauze adiționale, directiva Do/For va *partiționa iterațiile 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 aceleași 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 definiție, iar directiva PARALLEL DO/FOR acceptă toate clauzele directivei PARALLEL.

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

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

unde *tip* poate fi: STATIC, DYNAMIC, GUIDED sau RUNTIME, iar *chunksiz*e este o expresie întregă pozitivă, ce reflectă modul în care a fost divizat spațiul de iterații.

Cerințe pentru planificare statică

- dacă *size* nu este specificat, spațiul de iterație este divizat în părți egale, și o parte este asignată fiecărui thread (planificare pe block)
- dacă *size* este specificat, spațiul de iterație este divizat în părți, fiecare parte de un număr egal cu *chunksiz*e iterații, iar părțile sunt asignate în mod ciclic fiecărui thread (planificare tip block-cyclic)

Cerințe pentru planificare dinamică

- planificarea dinamică divide spațiul de iterație în părți de dimensiunea *size* și 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.

Cerințe pentru planificare ghidată (GUIDED)

- este similară celei dinamice, dar secvențele planificate sunt mari la început și apoi se micșorează exponențial.
- dimensiunea următoarei părți reprezintă (generic) numărul de iterații rămase, divizat la numărul thread-urilor.
- câmpul *size* specifică dimensiunea minimă a unei părți, 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.

Observații utile în alegerea unei anumite planificări sunt următoarele:

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

Standardul OpenMP propune și soluții 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 și 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 funcțiile ce permit identificarea threadurilor create și a numărului lor sunt utile funcțiile ce permit *alterarea dinamică* a numărului de threaduri *int omp_get_dynamic()*, respectiv activarea paralelismului imbricat *int omp_get_nested()*.

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

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

Directiva *Sections* acceptă clauzele PRIVATE, FIRSTPRIVATE și LASTPRIVATE, REDUCTION, NOWAIT fiecare secțiune trebuie însă să conțină un bloc structurat. La sfârșitul directivei o barieră implicită realizează sincronizarea ,în absența unei clauze NOWAIT. Forma scurtă a acestei directive este:

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

3.2.4.Directive de sincronizare și consistență a memoriei

Directiva SINGLE. Este necesar ca pentru anumite situații o anumită parte de procesare din regiunea paralelă să fie executată de *un singur thread*, fiind utilă pentru calcul de date globale, respectiv operații de intrare-ieșire. Primul thread care ajunge la directiva SINGLE va executa blocul, iar celelalte thread-uri vor aștepta până când blocul a fost executat. Directiva acceptă clauzele directivelor PRIVATE și FIRSTPRIVATE, în mod implicit la sfârșitul directivei se găsește o barieră. Această directivă este utilă pentru calcule de date globale sau operații 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 și își continuă execuția. 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 secvențial*. Deoarece directiva referă o execuție 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 secvența de cod astfel definită, doar atunci când threadurile anterioare au ieșit 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 și de către celelalte threaduri. Există o barieră implicită la sfârșitul directivelor DO/FOR, SECTIONS și 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 sfârșitul directivelor DO/FOR, SECTIONS și SINGLE (există anumite situații în care barierele sunt costisitoare). Directiva indică faptul că threadurile pot trece la instrucțiunea următoare fără a aștepta ca celelalte threaduri să termine execuția ciclului for. Sintaxa:

```
#pragma omp for nowait  
for loop
```

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

Directiva CRITICAL. O secțiune critică este un bloc de cod care poate fi executat de un singur thread la un moment dat și este folosită pentru a proteja variabilele partajate. Alături de funcțiile de bibliotecă specifice managementului lacătelor, directiva CRITICAL permite denumirea secțiunilor critice, astfel dacă un thread este într-o secțiune critică cu un nume dat, atunci nici un alt thread nu poate pătrunde într-o secțiune critică cu același nume. Dacă regiunea critică nu este numită secțiunea va primi un nume implicit, numele regiunilor critice este global. Regiunile critice reprezintă zone de serializare în cod ce trebuie reduse pentru creșterea performanței 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 șir pot necesita protecție 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 consistenței memoriei între threaduri diferite, astfel în acest mod întregul set de threaduri posedă o viziune consistentă a anumitor obiecte din memorie necesare aplicației. Dacă obiectele ce necesită sincronizarea sunt desemnate ca variabile, acestea pot fi specificate în lista opțională. 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 și ieșirea dintr-o secțiune CRITICAL sau ORDERED, PARALLEL, PARALLEL FOR și la sfârșitul directivelor

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

3.2.5. Date în aplicații OpenMP

Performanța programelor este în mod direct influențată 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 execuției regiunii paralele, cele mai importante fiind THREADPRIVATE ce definește domeniul de adresare ca fiind fișier, spațiu de nume respectiv static și un set de clauze ce permit controlul atributelor de partajare a variabilelor. Dacă o anumită variabilă este vizibilă la apariția unei construcții de partajare a lucrului și nu este specificată în lista THREADPRIVATE sau într-o clauză de attribute, atunci variabila este partajată.

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

Există situații în care este necesar ca anumite variabile să fie accesibile prin menținerea lor persistentă între regiuni paralele, *fără a fi copiate în spațiul 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 și sunt inițializate înainte de a fi accesate într-o regiune paralelă, ele fiind persistente. Utilizarea directivei presupune o serie de restricții între care, directiva trebuie să aibă scop la nivel de fișier sau de spațiu de nume, după toate declarațiile de variabile din *list* și înainte de orice referințe 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 funcționalității 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 ieșirea dintr-un ciclu. Sintaxa: **lastprivate(list)**

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

Clauza REDUCTION. O reducere produce o singură valoare din operații asociative precum: adunarea, înmulțirea, max, min, ȘI (logic), SAU (logic). Operația 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 execuția blocului structurat asociat construcției 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 sfârșitul unei directive SINGLE, fiind astfel cea mai folositoare pentru citirea în variabile private. Sintaxa: **#pragma omp single copyprivate(list)**

3.2.6. Legarea și imbricarea directivelor

Directivele PARALLEL permit crearea concurență de threaduri, iar directivele FOR și SECTIONS permit distribuirea taskurilor threadurilor create. În cazul în care nu există directiva PARALLEL, secvența de cod aferent directivelor SECTIONS și 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 și BARRIER se leagă de cea mai apropiată directivă PARALLEL.
- directiva ORDERED de leagă de cea mai apropiată directivă DO.

Paralelism imbricat (NESTED). Standardul oferă construcții 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 conține numai un thread, până la activarea paralelismului imbricat.
- directivele FOR, SECTIONS și SINGLE legate la aceeași directivă PARALLEL nu pot fi imbricate reciproc
- directivele CRITICAL cu același nume nu pot fi imbricate reciproc.

Directive orfane. Directivele sunt active în scopul *dinamic* (de execuție) 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 și blocurile COMMON sunt partajate, mai puțin în cazul în care sunt declarate cu THREADPRIVATE
- variabilele locale statice în C/C++ sunt partajate
- variabilele din lista de argumente moștenesc atributele cu scop de dată din rutina apelantă.
- toate celelalte variabile locale sunt private.

3.2.7. Funcții de bibliotecă pentru sincronizare

Deoarece sunt situații în care este necesară mai multă flexibilitate decât cea oferită de directivele CRITICAL și 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 inițializat înainte de a fi utilizat, și poate fi distrus când nu mai este necesar. Este posibilă utilizarea de *lacăte imbricate* ce pot fi utilizate multiplu de către același thread. Aceste funcții permit aceluiași thread să seteze un lacăt de mai multe ori, înainte de a-l reseta de același 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 consistența variabilei la o dată 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* și 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 și componentă distinctă de la adresa <https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019> și care va fi instalat pentru integrare în Visual Studio- IDE, cu scopul de a vizualiza variația gradului de paralelism pentru aplicație și 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 și un vector* folosind construcțiile de partajare a sarcinilor OpenMP și 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 și cum este ea implementată în OpenMP?

3.4.2. În modelul de date OpenMP, ce reprezintă FIRSTPRIVATE și 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 threadurilor, dar suportul pentru execuția unor clauze Parallel For imbricate?

3.5. Probleme propuse

3.5.1. Implementați un algoritm de înmulțire a două matrici pătratică folosind planificarea statică a threadurilor. Modificați soluția propusă pentru înmulțirea matricilor astfel încât folosind variabila de mediu OMP_NUM_THREADS să controlați numărul de threaduri, analizând totodată performanțele soluției la modificarea acestuia. Considerați următoarele cazuri de analiză: paralelizarea buclei exterioare, paralelizarea ambelor bucle, paralelizarea tuturor celor trei bucle.

3.5.2. Descrieți 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 parametri 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

Specificația *MPI (Message Passing Interface)* cunoaște diverse implementări (API-uri MPI) sub forma unor biblioteci ce conțin un set larg (peste 250) de rutine pentru schimbul de mesaje, managementul datelor și proceselor, rutine ce pot fi utilizate pentru un spectru variat de aplicații științifice implementabile eficient pe diverse sisteme paralele și/ sau distribuite.

4.1. Obiective

- Studiul bibliotecii de programare paralelă a sistemelor cu memorie distribuită bazată pe mesaje MPI
- Procese MPI și tipuri de comunicație
- 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 definește o *interfață de programare paralelă prin schimb explicit de mesaje*, ce poate fi utilizată pentru scrierea de aplicații 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 decompoziția datelor. MPI este o bibliotecă de funcții, ea definește nume de funcții, secvențe de apel și rezultatele pentru un set de funcții care pot fi apelate în programe C, C++ și 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 și biblioteca MPI.

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

- oferă o interfață de programare pentru aplicații paralele/distribuite
- oferă performanță, portabilitate și ușurință în folosire, semantica sa permite implementări diverse;
- semantica interfeței este independentă de limbaj (C, C++, Fortran)
- permite comunicație eficientă *punct-la-punct și punct-multipunct* în medii eterogene (rețele diverse);
- conține un set de rutine pentru lucrul cu *grupuri de procese*, permite scheme de adresare complexe.
- permite *comunicație colectivă și operații de calcul global*, ce conferă scalabilitate sistemului
- oferă contexte de comunicare și comunicatori pentru dezvoltarea de biblioteci de funcții necesare programării paralele;
- conține rutine pentru definirea de topologii virtuale de grup;
- oferă simplitate dar și completitudine bazat pe un număr extrem de mare de funcții, însă aplicații relativ complexe pot fi programate folosind un *set minimal de 6 funcții esențiale*.

4.2.1. Modelul de comunicație

Soluția 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 biți, astfel structura unui mesaj devine *adresa, contor, tip_date*. Termenul de *buffer de mesaj* poate fi utilizat cu semnificații multiple, astfel:

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

Sincron vs asincron. Mesajul poate fi transferat direct între bufferele declarate în aplicația utilizator, sau pentru comunicație asincronă, mesajul este *stocat temporar* într-un buffer creat dinamic de sistem înaintea depozitării sale în bufferul receptor, respectiv utilizatorul poate declara inițial un bufer suficient de mare astfel încât să permită transferul oricărui mesaj ce necesită stocare intermediară și utilizare în operația de transfer. Comunicare prin transfer sincron de mesaje definește 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-recepției de mesaje se încheie doar atunci când mesajul transmis este și recepționat. Avantajele pe care le prezintă referă semantica simplă și ușor de utilizat. Un proces care revine din subrutina send va ști cu certitudine că mesajul trimis a fost recepționat. De asemenea un avantaj important îl constituie faptul că nici sistemul și nici utilizatorul nu trebuie să mențină un buffer pentru mesaj, acesta poate fi copiat direct în spațiul de adrese al procesului receptor. Datorită simplității, un număr mare de sisteme paralele implementează diferite variante ale acestui mod de comunicare.

Comunicarea bazată pe emisie - recepție blocantă este caracterizată de transmiterea unui mesaj în mod blocant, executată *fără a aștepta recepția acestuia* de către procesul destinație, astfel este posibilă încheierea execuției unei operații de transmisie chiar înaintea începerii execuției operației de recepție corespunzătoare. Tipurile de comunicare sunt prezentate în tabelul 1.

| Evenimentul de comunicație | Sincron | Blocant | Nonblocant |
|--------------------------------------|-------------------|----------------|-------------------------------|
| Revenirea din send este indicată | Mesaj recepționat | Mesaj transmis | Transmitere de mesaj inițiată |
| Bufferare mesaj | Nu e necesară | E necesară | E necesară |
| Verificarea stării | Nu e necesară | Nu e necesară | E necesară |
| Încărcare generată de așteptare | Mare | Medie | Mică |
| Suprapunere comunicație și procesare | Nu | Da | Da |

Tabel 4.1. Moduri de comunicare aflate la baza primitivelor MPI

Comunicare bazată pe emisie - recepție neblocantă. Transmisia neblocantă se încheie *imediat după ce a fost notificat sistemul* de existența unui mesaj de transmis, respectiv recepționat, mesaj care poate fi în tranzit, deja sosit sau chiar netransmis încă. Pentru această variantă, sistemul trebuie să implementeze un *buffer temporar*. Implementările performante conțin funcții specializate pentru verificarea *stării bufferului* sau pentru implementarea unor mecanisme de așteptare până 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 aplicației.

Avantajele oferite de comunicația neblocantă pot fi exploatate în contextul utilizării suportului hardware de comunicație dedicat, astfel încărcarea indusă de comunicație poate fi aproape integral mascată prin operații neblocante. Operațiile neblocante pot fi utilizate și cu un protocol bufferat, astfel emițătorul inițiază operația DMA și revine, însă datele sunt disponibile accesului (sunt sigure) doar după ce operația 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 și recepționate. MPI asigură un set deosebit de complex de tipuri de date predefinite, set care include toate tipurile de bază din C, C++ și Fortran, alături de două tipuri de date specifice MPI: MPI_BYTE și MPI_PACKED.

Context de comunicație. 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 și grup de procese. Procesul MPI este unitatea fundamentală de calcul, fiind constituit dintr-un *fir de control independent* și un *spațiu de adrese privat*, astfel un proces nu poate accesa direct variabilele definite în spațiul 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 și 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 dificultăți 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, apartenența la grup este statică însă suportă atașarea-detașarea 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 și n-1, unde n este numărul proceselor din grup. Inițializarea unei aplicații MPI folosind rutina de inițializare MPI_INIT, creează un singur grup, numit MPI_COMM_WORLD, care include toate procesele MPI ce formează aplicația.

Procesele pot utiliza *comunicație punct-la-punct* cu scopul transmisiei- recepționării de mesaje utilă implementării comunicațiilor locale nestructurate. De asemenea un grup de procese poate apela operații de tip *comunicații colective* pentru a simplifica implementarea unor operații globale, deși standardul nu oferă în mod explicit suport pentru multithreading, unele implementări includ mecanisme de acest fel.

Comunicatorul integrează informațiile de context și informațiile de identificare a grupului. MPI_INIT definește pentru fiecare proces apelat comunicatorul implicit MPI_COMM_WORLD. Toate comunicațiile 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 și poate fi folosit pentru specificarea sursei sau destinației unui mesaj. Folosind MPI_COMM_WORLD, fiecare proces poate comunica cu oricare altul, iar grupul MPI_COMM_WORLD reprezintă mulțimea tuturor proceselor MPI. Informațiile specifice comunicatorului pot fi accesibile prin funcțiile:

MPI_COMM_RANK (MPI_comm comm, int *rank)

Funcția MPI_COMM_RANK returnează rangul procesului apelant în grupul comunicatorului *comm*.

MPI_COMM_SIZE (MPI_comm comm, int *size)

Funcția MPI_COMM_SIZE returnează în variabila *size* numărul de procese din grupul asociat comunicatorului *comm*. Fiecare proces aparținând unui comunicatori este identificat prin rangul său.

4.2.2. Modelul computațional

Stilul de programare care poate fi folosit cu MPI se bazează fie pe modelul **MIMD** (Multiple instructions multiple data) ce presupune existența 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 același program, dar execută porțiuni diferite de cod selectate pe baza unor instrucțiuni de condiție. Standardul MPI a fost proiectat astfel încât programatorul începător să nu simtă complexitatea de la primele programe. Programarea unei aplicații minimale MPI se poate face

cunoscând doar un set minimal de 6 funcții ce reprezintă nucleul suficient pentru scrierea unei aplicații:

MPI_Init (int *argc, char**argv) este funcția care inițializează variabilele locale MPI și înștiințează sistemul global MPI despre execuția unui nou proces MPI. Funcția acționează și ca barieră de inițializare, în sensul ca nici un proces MPI nu-și poate continua execuția până când toate celelalte procese nu au executat partea lor de inițializare, 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() întoarce numărul de procese care fac parte din grupul specificat ca parametru

MPI_Send() și **MPI_Recv()** sunt funcțiile de bază care implementează comunicația prin mesaje între procese.

MPI_Finalize() pentru anunțarea sistemului MPI despre terminarea procesului MPI cu succes (valoarea lui ierr=0), această rutină este apelată de fiecare proces MPI. După această funcție nici o altă funcție nu poate fi apelată, toate comunicațiile în așteptare trebuie încheiate înaintea apelului ei.

Apelul funcțiilor de bibliotecă necesită includerea antetului **mpi.h** ce conține constantele referite în rutinele MPI (tipuri de date, comunicatori, erori) și interfețele funcțiilor MPI. O operație este considerată *terminată local* în cadrul unui anumit proces, dacă acesta a terminat partea sa ,atât în comunicația punct-la-punct, cât și în cea colectivă. O operație de comunicare este *terminată global*, dacă toate procesele implicate în comunicare au *terminat local* părțile ce le revin din aceasta.

4.2.3. Modul de comunicație punct-la-punct

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

Programele paralele necesită implementarea unor soluții algoritmice capabile să minimizeze timpul de răspuns, iar MPI oferă propriile mecanisme pentru instrumentarea codului astfel încât să permită analiza performanței. Funcția **MPI_Wtime**, returnează o valoare dublă precizie a numărului de secunde raportat la un anumit moment de timp trecut, garantat nemodificabil pe durata execuției programului, fapt ce permite inserarea diverselor apeluri ale rutinei în codul sursă pentru măsurarea unor intervale de timp de execuție. Funcțiile bibliotecii MPI utilizate în analiza performanțelor bazat pe elementul timp de procesare sunt **MPI_Wtime(void)**, **MPI_Wtick(void)**. Lista parametrilor utilizați în rutinele MPI și semnificația 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 conține mesajul(transmis-receptionat) |
| count | Număr de articole specificate în bufer (număr maxim de articole recepționate) |
| datatype | Tipul de dată a fiecărui element transmis |

| parametru | Semnificație |
|-----------|---|
| Root | Rangul unui proces specific (sursa sau destinația pot fi identice) |
| Size | Numărul de procese din grup |
| Tag | Tagul de mesaj transmis-recepționat |
| source | Rangul procesului sursă în comunicator |
| dest | Rangul procesului destinație în comunicator |
| status | Tablou de întregi ce conține starea, referă o modul de recepție a mesajului (sursa, tagul și codul de eroare) |

Tabel 4.2. Coduri de eroare și parametrii rutinelor MPI

Pentru a putea construi o aplicație paralelă performantă este necesar un control performant asupra transmiterii mesajelor. În acest sens, MPI definește patru moduri de comunicare: **standard, sincron, bufferat și ready** (vezi tabelul 4.3.). Funcția de comunicare se obține prin combinarea modului de comunicație cu un mod de blocare, moduri ce vor fi detaliate la secțiunea funcții pentru comunicație. Toate modurile de comunicație există atât în forma blocantă cât și neblocantă, în forma blocantă, întoarcerea dintr-o funcție implică terminarea ei cu succes MPI nu impune restricții asupra modului de potrivire a funcțiilor de transmisie cu cele de recepție, existând astfel mai multe posibile combinații a căror funcționare corectă intră însă în sarcina programatorului. MPI definește două moduri de comunicare:

- fără blocare (imediat), caz în care funcțiile de transmisie sau recepție se termină înainte ca informația să fie trimisă/recepționată, iar bufferul indicat într-o funcție neblocantă nu trebuie folosit până la detectarea sfârșitului funcției inițiate.
- cu blocare, caz în care după terminarea funcției, bufferul de transmisie sau recepție poate fi refolosit.

| Tip | Descriere |
|---------------|--|
| Send sincron | operația 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 operația de trimitere să se termine întotdeauna înainte ca mesajul să fie recepționat la destinație. |
| Send standard | operația de trimitere poate începe chiar dacă operația corespunzătoare de recepție nu a început. |
| Send ready | operația send poate începe doar dacă operația corespunzătoare de recepție 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 informații referitoare la operația MPI_recv, aceasta este o structură de date cu trei câmpuri: sursa mesajului, tagul și cod de eroare, iar lungimea mesajului transmis poate fi verificată cu ajutorul funcției 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 destinație, 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* conținute de *buffer*

dest reprezintă procesul destinație specificată prin intermediul rangului din cadrul comunicatorului *comm*;

tag este un marcator la dispoziția programatorului pentru a distinge între diferite tipuri de mesaje;

Transmisie sincronă (Synchronous send). *Synchronous send* se folosește atunci când se dorește confirmarea primirii mesajului, astfel destinația trimite spre sursă o confirmare iar transmisia este considerată încheiată numai după primirea confirmării. Primitiva *send* se consideră încheiată atunci când o operație *receive* a fost lansată, astfel bufferul aplicației este disponibil pentru a recepționa mesajul. Este posibilă o transmisie sincronă neblocantă ce nu presupune transmiterea mesajului, ci doar inexistența bufferării suplimentare la receptor, dar un buffer sistem la emițător poate fi necesar, iar pentru a elimina copii multiple ale mesajului se poate utiliza emisia blocantă. Această formă de transfer se folosește atunci când este nevoie de siguranță 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* folosește un buffer pentru trimiterea mesajului, dacă acesta nu poate fi trimis la momentul dorit. Avantajul față de metoda clasică îl constituie predictibilitatea recepției mesajului. Atașarea unui buffer se realizează cu ajutorul funcției:

```
int MPI_BUFFER_ATTACH (void *buffer, int size)
```

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

```
int MPI_BUFFER_DETACH (void *buffer, int size)
```

Observații:

- după detașare, utilizatorul poate reutiliza sau dealoca spațiul ocupat de tampon
- unele operații, de atașare și detașare, 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** (și nu *void** respectiv *void***) pentru a evita conversii forțate de tip (*cast*).

Transmisie ready (Ready send). Formele *Ready send*, ca și *buffered send* se termină imediat, iar comunicarea este garantată a se efectua cu succes dacă o operație *receive* corespunzătoare a fost apelată. La expedierea mesajului, procesul expeditor pune mesajul în comunicator, sperând ca destinatarul să îl aștepte și 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)
```

Recepție 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 și 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.

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

4.2.4. Comunicații colective, comunicatori și topologii virtuale

Mecanismele abstracte folosite în comunicație sunt : comunicatori, context de comunicare, topologii virtuale. În multe aplicații paralele este necesar ca operațiile de comunicație să fie restricționate la un set limitat de procese. MPI definește o serie de concepte și introduce diverse mecanisme pentru lucrul cu procese structurate în grupuri și organizate folosind comunicatorii. MPI introduce noțiunea de *comunicator* pentru a defini cadrul unei comunicații colective, în particular procesele care sunt implicate în această comunicare, dar și pentru a permite construcția modulară a programelor paralele în mod: secvențial, paralel sau-și concurrent.

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

Comunicația inter-grupuri este, de obicei, gestionată de un conducător al grupului. Grupurile pot fi create și prin intermediul reuniunii, intersecției, diferenței între grupuri deja existente. Noțiunea de comunicator este fundamentală pentru definirea într-un mod modular a bibliotecilor de funcții care pot fi invocate de procesele utilizator. Putându-și defini propriul comunicator (alocat de către sistem, cu asigurarea unicității), procesele sistem pot schimba mesaje fără a interfera cu comunicația dintre procesele aplicației utilizatorului.

Este necesară distincția clară între diferitele tipuri de comunicatori. Astfel un *comunicator* este un obiect cu atribute și reguli pentru creare, utilizare și 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 și 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 toți comunicatorii sunt intracomunicatori și au aceleași atribute.

Unui comunicator *i* se pot atașa informații adiționale alături de cele de grup și context, informație numită **topologie**. Topologia reprezintă un mecanism ce realizează asocierea diverselor *scheme de adresare proceselor* aflate într-un grup. Grupurile și 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 funcțiile 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 aplicații, 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 distincție între topologia virtuală a proceselor și topologia reală a sistemului pe care acestea sunt executate. Topologia virtuală poate fi exploatată de sistem în plasarea proceselor pe procesoare, pentru a ameliora performanțele.

Trebuie reținut faptul că topologia virtuală reflectă caracteristicile aplicației și poate fi folosită în *îmbunătățirea performanțelor*. Sunt cunoscute tehnici standard de mapare a unor topologii grilă / tor pe hipercuburi sau grile de procesoare. Pentru topologiile graf metodele sunt mai complexe și se bazează adesea pe euristici. În afara informației furnizate algoritmului de plasare a proceselor pe procesoare, topologiile virtuale au ca rol facilitarea scrierii programelor, într-o formă mai ușor de înțeles. 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 și 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 considerați de maparea grupului de procese pe arhitectura fizică. Noțiunea de topologie virtuală este deosebit de utilă, atunci când nu este disponibil un mecanism eficient de mapare a topologiei virtuale pe un calculatormasiv paralel. Astfel, dacă procesele utilizatorului vor comunica în principal cu cei mai apropiați vecini după modelul unui grid bidimensional, se va putea crea o topologie virtuală care să reflecte acest fapt. În acest mod este realizat accesul la rutinele convenabile, de exemplu, calculând rangul oricărui proces dat prin coordonatele sale în grid, luând în considerare condițiile limită și returnând MPI_NULL_PROC dacă se iese din grid. În particular există rutine care calculează rangurile celor mai apropiați vecini, rang ce poate fi folosit ca argument pentru primitivele de transmisie /recepție MPI_SEND, MPI_RECV, MPI_SENDRECV etc. Deși o topologie virtuală scoate în evidență șabloanele 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, absența 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. Deși specificarea topologiei în termeni de graf este foarte generală, în multe aplicații specificarea unei topologii mai simple, regulate, este mai convenabilă. De exemplu înel, grile bi- sau tri-dimensionale, tor. Aceste topologii sunt complet definite prin numărul de dimensiuni și 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 carteziane, 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 carteziană dacă fiecare proces este conectat cu vecinii săi printr-un grid virtual. Mecanismul utilizat de MPI pentru a asigura ranguri proceselor dintr-un anumit domeniu de comunicație nu utilizează informație despre rețeaua de comunicație, ci oferă un set de funcții ce permit programatorului *aranjarea proceselor în diverse topologii fără a specifica explicit maparea lor la procesoare*, responsabilitatea unei mapări capabile să reducă costurile transmisiei de mesaje este în totalitate a bibliotecii MPI.

Comunicația colectivă permite difuzarea datelor către toate procesele unui grup, sincronizarea la bariere ca mecanism de control al execuției, distribuția colectarea datelor de la procesele unui grup, operații globale de reducere pe grupuri de procese (suma, minim, maxim) sau comunicație de tip all-to-all. Toate aceste operații se pot împărți în două mari clase: *operații de transfer de date și operații de calcul global*.

Deși operațiile de comunicație colectivă nu acționează 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 operații de sincronizare globală înainte respectiv după apelul funcției colective. Deoarece operațiile colective sunt virtual sincrone, ele *nu necesită marcajul de mesaj*(tag-ul). Pentru cele mai multe operații colective există două variante astfel: transfer de date de aceeași 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) și *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 operații *all-to-all*, în care fiecare proces trimite date, respectiv recepționează date de la celelalte procese, prin utilizarea unei singure funcții cu semnătură complexă.

Operațiile de calcul global pot fi împărțite în operațiile de reducere și operații de scanare. Pentru operațiile de reducere funcția de reducere se aplică pe toate datele corespunzătoare fiecărui proces din grup, iar pentru operațiile de scanare, funcția de reducere se aplică pe datele proceselor cu rang mai mic decât cel al procesului care execută operația.

Funcția de reducere trebuie să fie asociativă și comutativă, iar rezultatul unei operații globale poate fi cunoscut de *toate procesele sau doar de un singur proces*. În operațiile globale pot fi folosite atât operațiile predefinite în MPI cât și operații definite de utilizator. În cadrul comunicației colective, nu se folosesc etichete de mesaj, coordonarea comunicației proceselor este implicită și toate rutinele de comunicație colectivă se blochează până când comunicația este terminată local.

Operații colective. Comunicarea colectivă implică un grup de procese între membrii căruia se desfășoară comunicația. Biblioteca MPI suportă comunicații colective cum ar fi difuzarea, *dispersarea/adunarea* (scatter/gather), *schimb total de date*, *agregare și 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 comunicațiilor colective:

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

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

Unic argument este comunicatorul *comm* ce definește grupul de procese sincronizate, astfel procesul apelant se blochează până când toți membrii grupului execută apelul funcției (ating bariera).

Transmisia broadcast. În transmisia *one –to-all*, procesul root transmite același mesaj stocat în bufferul *buffer* (un număr de *count* de intrări de date având tipul *datatype*) tuturor proceselor inclusiv lui însuși, procese incluse în comunicatorul *comm*. Toate procesele implicate în comunicație trebuie să specifice aceeași 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.

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

Funcția Gather. Procesul țintă (poate fi chiar rădăcina pentru topologii de tip arbore) recepționează mesaje personalizate de la fiecare din cele n procese (inclusiv de la el însuși). Cele n mesaje sunt *concatenate în ordinea rangurilor proceselor* și stocate în bufferul de recepție al procesului receptor. Fiecare buffer emițător este identificat de tripletul *sendbuf, sendcount, sendtype*, iar bufferul de recepție este ignorat pentru procesele nonroot, iar pentru procesul root este identificat de tripletul *recvbuf, recvcount, recvtype*.

Funcția implementează comunicația de tip *all-to-one*, toate procesele transferă datele aflate în *inbuf* procesului rădăcină, care plasează datele în locații 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 recepție
- *target* este rangul procesului receptor

Funcția Scatter descrie operația inversă operației 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 funcțiilor de comunicație colectivă se vor consulta resursele [1][3].

Pentru *analiza de performanță* și instrumentarea codului aplicației cu scopul de a identifica timpul necesar procesării este utilă funcția *MPI_wtime()* ce returnează valoarea timpului raportat la ceasul sistem. Deasemenea este util Analizorul de concurență pentru Visual Studio care poate fi descărcat ca și componentă distinctă de la adresa <https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019> și care va fi instalat pentru integrare în Visual Studio- IDE, cu scopul de a vizualiza variația gradului de paralelism pentru fiecare aplicație și consumul de memorie la execuția codului.

4.3.Exemple

4.3.1. Realizați o transformare a programului secvențial pentru calculul lui PI bazat pe integrarea numerică a funcției $f(x)=4/(1-x^2)$ în intervalul $[0, 1]$. Să se ofere o soluție de paralelizare prin împărțirea intervalului de integrare aferent numărului de procesoare din sistem. Pentru algoritm se poate folosi aproximarea funcției cu sumarea $k=1..n$ $(4/(1+((k-.5)/n)^2))$ și fiecare nod va primi numărul de dreptunghiuri folosite în aproximare, va calcula aria dreptunghiului și apoi se sincronizează pentru calculul sumei globale. Implementați 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 aceeași (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 equally 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=3.141592653589793238462643;
    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 aplicație simplă folosind transmiterea de mesaje în accepțiunea în care se definesc un proces receptor și n-1 procese emițător. Acestea își transmit identificatorul de proces și numele hostului procesului receptor care le tipărește.

4.5.2. Să se implementeze un program în care sunt definite un număr impar de procese. Procesul zero inițializează un tablou de întregi și distribuie tabloul tuturor proceselor folosind primitiva Scatter. Procesele receptor își primesc părțile corespunzătoare din tabloul sursă. Fiecare proces răspunde cu un mesaj ce conține numele hostului pe care procesul se execută și partea sa din tabloul distribuit anterior, iar procesul rădăcină recepționează și tipărește mesajele.

4.5.3. Să se calculeze produsul a doi vectori (x, y de dimensiune n) distribuiți unui grup de procese. Tablourile ce conțin vectorii sunt alocate static, iar dimensiunea vectorilor este divizibilă cu p numărul de procesoare din sistem. Propuneți două soluții de implementare pentru operația de reducere și analizați modul de execuție și performanța folosind analizorul de concurență ce poate fi descărcat de la adresa <http://msdn.microsoft.com/en-us/library/dd537632.aspx> și integrat în Visual Studio.

4.5.4. Implementați un program care să determine timpul necesar execuției operației MPI_Barrier. Utilizați comunicatorul implicit MPI_COMM_WORLD. Tipăriți dimensiunea comunicatorului

asigurându-vă ca atât emițătorul cât și receptorul sunt pregătiți la începutul testului. Cum variază performanța primitivei variind dimensiunea comunicatorului?

4.5.5. Propuneți un program care să realizeze recepția corectă a mesajelor de la toate procesele emițătoare, astfel încât toate cele 100 procese să transmită procesului 0, utilizând recepția nonblocantă și primitiva `Wait_some`.

4.5.6. Modificați problema conectării procesoarelor unei mașini paralele într-o topologie de tip inel înlocuind referirea proceselor vecine folosind rangurile rank-1 și rank+1 folosind rutinele MPI specifice topologice (topologii catreziene, liniare) utile pentru “ lumi ” de procese ce necesită comunicatori distincti.

4.6. Miniproiect

Să se descrie în pseudocod și să se implementeze și evalueze performanța implementării pentru soluții paralele ale algoritmilor de înmulțire matrici optimizați 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 aplicației 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 și execuție*
<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 și o interfață de programare a aplicațiilor dezvoltată cu scopul de a accelera operațiile de calcul masiv pe date multiple prin folosirea puterii de calcul disponibilă în procesoarele grafice GPU. Această platformă permite dezvoltatorilor și 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 și probleme generale de procesare. În execuție, o arhitectură de tip GPU folosește 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 și 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 performanță

5.2. Concepte

Plăcile grafice au devenit atât de puternice încât sunt folosite pentru calcul matematic divers, cum ar fi operații complexe cu matrici, vectori operații necesare pentru diverse simulări vizuale și fizice complexe în diverse domenii cum ar fi :industria auto, prelucrări video și de imagini, criptografie, design electronic, simulări fizice diverse, prelucrări multimedia. Unitatile tip GPU sunt potrivite pentru paralelismul de date, intensiv computationally. Datorită faptului că aceleași instrucțiuni 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 computationally, latența 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 susținut această tendință prin lansarea bibliotecii CUDA (Compute Unified Device Architecture), pentru a permite dezvoltatorilor de aplicații să scrie cod care poate fi încărcat pe o placă de tip NVIDIA pentru a fi executat de GPU-urile acesteia. Multe aplicații 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 și arce sunt mapate către threaduri paralele.

Spre deosebire de CPU, GPU-urile sunt optimizate pentru calcul paralel pe multiple seturi de date, diferența majoră între arhitectura CPU și cea a GPU constă în modalitatea de acces la memorie: CPU-ul folosește între 1 și 3 canale de memorie de câte 64 biți lățime fiecare, în timp ce GPU-urile actuale pot folosi până la 8 canale de memorie paralele, de 64 biți lățime fiecare. nVidia a identificat acest potențial al GPU-urilor și a proiectat o interfață de programare prin care se permite anumitor programe accesul la această putere de calcul. Acest model a fost implementat în toate GPU-urile începând cu seria 8800, fiind suportat în principal pe GPU-urile NVIDIA construite pe arhitectura Tesla, iar plăcile grafice care susțin execuția aplicațiilor ce folosesc biblioteca CUDA sunt seriile GeForce 8, Quadro, și Tesla.

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

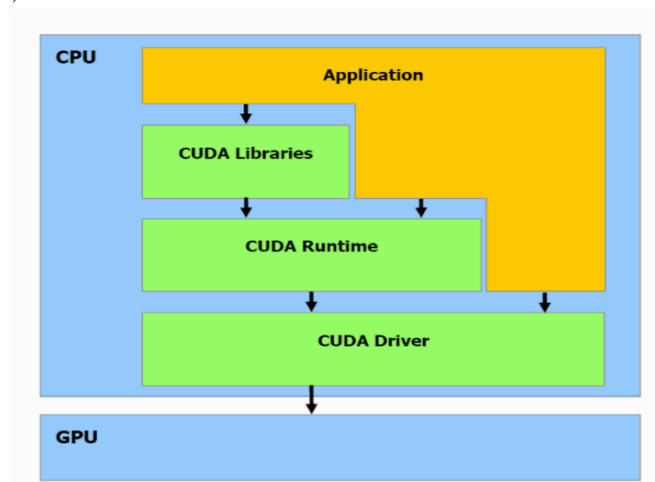


Figura 5.1. Stiva software CUDA [2]

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 să 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

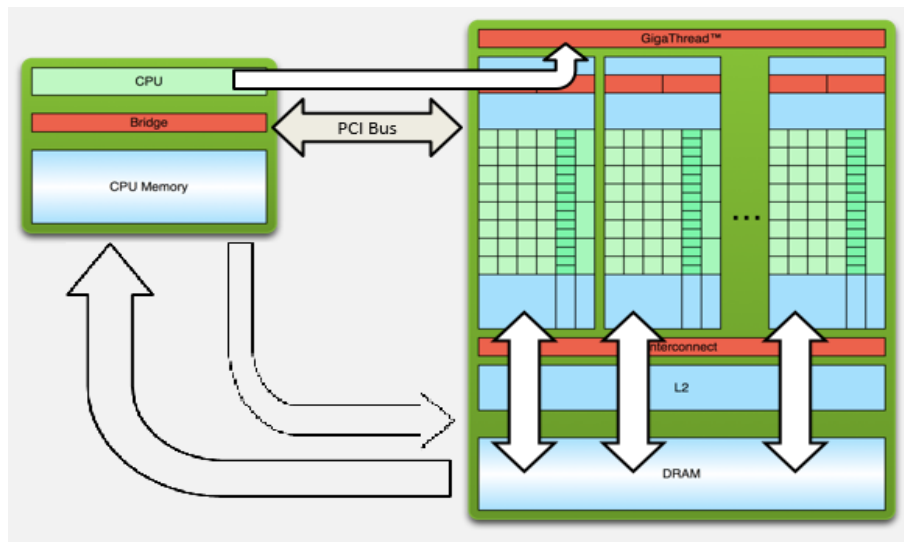


Figura 5.2. Flux de procesare CUDA [2]

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 simplu 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 copiiile 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 înapoi 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 dispozitive multiple |
| | Memoria: liniară sau tablouri CUDA |
| | Interoperabilitate OpenGL și DirectX |
| | Asincronicitate: funcția <code>__global__</code> 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 atomice (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 : <code>int2 i = make int2(i, j)</code> |
| | 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 concurrent 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. În 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);
```

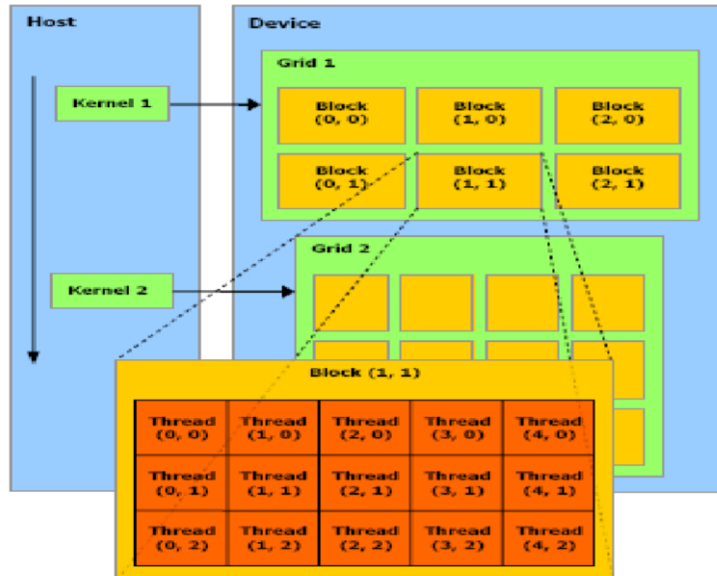


Figura 5.3. Structuri de threaduri

5.2.3. Modelul de memorie și 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.

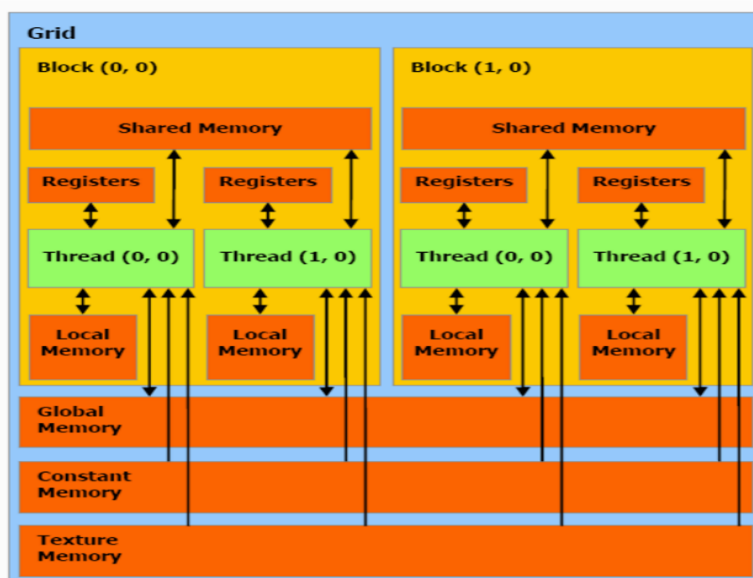


Figura 5.4. Modelul de memorie [2]

Toate multiprocesoarele accesează o memorie mare device globală atât pentru operații gather cât și pentru operații scatter, memorie care este relativ încetă 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 în 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.

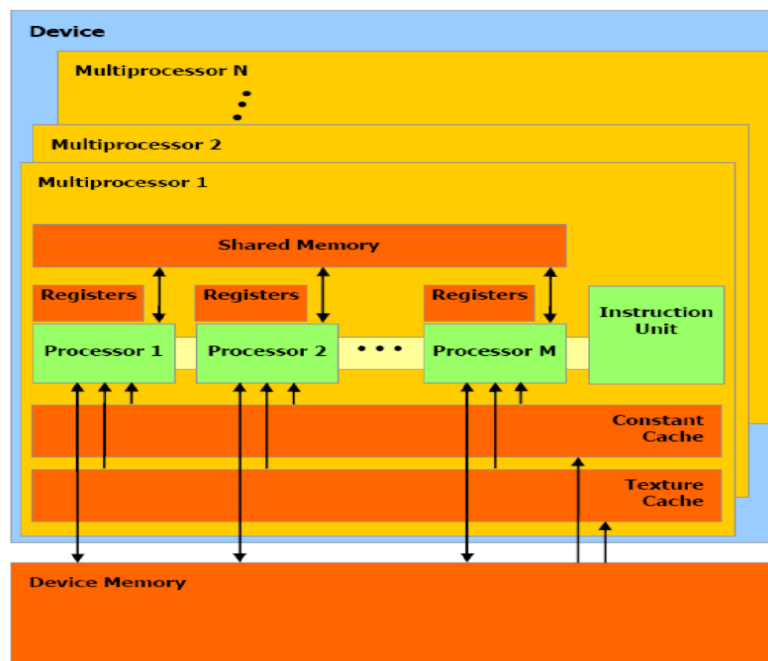


Figura 5.5. Modelul hardware [2]

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 accesăm memoria impactează foarte mult performanța sistemului. Cum putem avea arhitecturi foarte diferite din punct de vedere al ierarhiei de memorie este important de înțeles că nu putem dezvolta un program care să ruleze optim în toate cazurile. Un program CUDA este portabil și astfel poate fi ușor rulat pe diferite arhitecturi NVIDIA CUDA, însă de cele mai multe ori trebuie ajustat în funcție de arhitectură pentru o performanță optimă. În general pentru arhitecturile de tip GPU, memoria locală este împărțită în module de SRAM identice, denumite bancuri de memorie (memory banks). Fiecare banc conține o valoare succesivă de 32 biți (de exemplu, un int sau un float), astfel încât accesul consecutiv într-un array provenite de la threaduri consecutive să fie foarte rapid. Conflicte au loc atunci când se fac cereri multiple asupra datelor aflate în același banc de memorie.

Conflictele de acces la bancuri de memorie (cache) pot reduce semnificativ performanța. Când 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. În unele cazuri, dacă toate threadurile citească aceeași adresă 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 oricâte 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 procesor, de aceea, este garantată execuția unui bloc thread pe un singur procesor.

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 funcție de dimensiunea 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 apelează două kerneluri în succesiune, al doilea va fi lansat fără să mai aștepte 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 și numai după ce primul kernel este terminat se va executa cel de al doilea.

5.2.6. Tratarea erorilor

Toate funcțiile 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 eșua.
- 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 după executarea acestuia prin apelarea metodei `cudaGetLastError()`:

Exemplu:

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

5.2.7. Gestionarea dispozitivelor

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

- Aplicația poate interoga și 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 opțional împreună cu `__device__`. Dacă o variabilă este declarată numai cu calificatorul `__device__` atunci această variabilă se află în memoria globală și are aceeași durată 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 durată 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 durată 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 următoarele soluții:

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

O altă problemă o constituie accesul la memorie. Întrucât accesul 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 disproporționați, 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 aplicație, parte a NVIDIA CUDA Toolkit, care poate să ofere multe informații cu privire la problemele de performanță dintr-o aplicație CUDA. Se crează o sesiune și 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 |
|--|----------------|
| <pre>const int N = 1024; const int blocksize = 16;</pre> | Set grid size |
| <pre>__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]; }</pre> | 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 Concurrency 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%.01f", 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%.01f", 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\n" );
for (i=0; i<N*D; i++){
    for (j=0; j<N*D; j++){
        printf("\t%.01f", 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\n" );
    for (i=0; i<N*D; i++){
        for (j=0; j<N*D; j++){
            printf("\t%.01f", 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. Prezența 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 algoritmi 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 pot executa în paralel).

5.5.2. Implementați utilizând CUDA funcția `matrix_multiply_simple` care va realiza înmulțirea a două matrice primite ca parametru. Realizați o înmulțire 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 partajată între 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` și `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. Bibliotecă 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 în vedere faptul că 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 furnizează un flux sigur de date între două 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 adresă 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 legătură 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 concurrent mai multe procese care au stabilite conexiuni în rețea, așteptând 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 între 0 și 65535 (deoarece sunt numere reprezentate pe 16 biți), numerele cuprinse între 0 și 1023 fiind însă rezervate unor servicii sistem și din acest motiv nu trebuie folosite în aplicații.

Pentru implementarea aplicațiilor de rețea pachetul *Java.net* oferă suport independent de sistem, având la baza modelului comunicația de tip client - server. Acest pachet conține clase, interfețe și excepții ce implementează în formă obiectuală conceptele uzuale de comunicare. Astfel, un client Java, pentru a obține un anumit serviciu de la un server remote, folosește adresa serverului pentru a solicita conexiunea, iar serverul ascultă rețeaua și așteaptă cereri din partea clienților. Stabilirea conexiunii presupune utilizarea ca și identificatori de hosturi comunicante a adresei IP și a adresei portului.

6.2.1. Elemente în programarea cu socketuri

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

Metodele statice ale clasei sunt:

- *Public static InetAddress getByName(String host) throws UnknownHostException* returnează numele hostului local pentru care se cunoaște adresa IP.
- *Public static InetAddress getLocalhost() throws UnknownHostException* -determină adresa IP a hostului local
- *Public static InetAddress getAllByName(String host)* - returnează în tablou adresele IP ale hostului curent
- *Public byte getAddress()*, returnează adresa IP a obiectului curent în formă de șir
- *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 facilități necesare manipulării locațiilor universale 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 funcție de tipul resursei. Clasa care permite lucrul cu URL-uri este *java.net.URL*. Aceasta are mai mulți constructori pentru crearea de obiecte ce reprezintă referințe către resurse aflate în rețea, cel mai uzual fiind cel care primește ca parametru un șir de caractere. În cazul în care șirul nu reprezintă un URL valid va fi aruncată o excepție de tipul *MalformedURLException*.

Constructorii clasei sunt:

- *Public URL (String protocol, String host, int port, String file) throws MalformedURLException* – creează un obiect URL
- *Public URL (String protocol, String host, String file) throws MalformedURLException*, creează un nou obiect URL absolut pentru care se folosește 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 parțială componentele lipsă vor fi preluate din context.
- *Public URL (String spec) throws MalformedURLException* – creează un obiect URL din reprezentarea string dată.

Un obiect de tip URL poate fi folosit pentru:

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

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

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

Operațiunea de trimitere de date dintr-un program către un URL este similară cu trimiterea de date dintr-un formular de tip FORM aflat într-o pagină HTML. Metoda folosită pentru trimitere este POST. În 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 două procese ce rulează într-o rețea. Fiecare socket este atașat unui port astfel încât să poată identifica unic procesul (aplicația) căruia îi sunt destinate datele.

Socket-urile sunt de două tipuri:

1. TCP, implementate de clasele *Socket* și *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. În 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 inițiază conversația cu un server, solicitând un anumit serviciu. Uzual, un server trebuie să 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.

Începând cu versiunea 1.4 a platformei standard Java, există o clasă utilitară care implementează o pereche de tipul (adresa IP, număr port). Aceasta este *InetSocketAddress* (derivată din *SocketAddress*), obiectele sale fiind utilizate de constructori și metode definite în cadrul claselor ce descriu socketuri, pentru a specifica cei doi parametri necesari identificării unui proces care trimite sau recepționează 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 aștepta conexiuni de la clienți. Un socket în Java este reprezentarea unei conexiuni TCP, folosind această clasă un client poate crea canale de comunicație de tip streamuri de date. După stabilirea conexiunii, hosturile locale, respectiv remote pot comunica folosind o conexiune tip full duplex.

Constructorii cei mai utilizați ai acestei clase sunt:

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

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

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

streamurilor utilizate în comunicația bidirecțională. Comunicația folosind socketurile în Java presupune crearea unui Socket apoi utilizarea metodelor *getInputStream()* și *getOutputStream()* pentru a obține streamurile necesare comunicației. Alte metode definite în această clasă sunt: *getInetAddress()*, *getPort()*, *toString()*. Excepțiile sunt cele cauzate de imposibilitatea legării la hostul local sau remote, refuzarea unei conexiuni datorită inexistenței serverului care să asculte pe un anumit port sau imposibilitatea atingerii hostului remote.

Clasa ServerSocket. Această clasă oferă funcționalitatea mecanismului prin care un server *acceptă conexiuni* de la clienți, ea permite crearea unei conexiuni socket pentru fiecare client apoi serverul va gestiona aceste conexiuni extrăgând streamuri de intrare și ieșire 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 viață al unui server clasic conține 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 și out), clientul și serverul vor continua interacțiunea respectând un anumit protocol, apoi se va închide conexiunea de către unul din capetele comunicante și în final se va aștepta 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 și cu numărul de cereri de conectare care pot fi păstrate în așteptare (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, permițând alegerea acesteia.

Complementar există definiți constructori care pot fi utilizați 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 soluția de comunicație 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 și adresa locală sunt încă accesibile folosind metodele *GetInetAddress()*, *getLocalPort()*, *getPort()*.

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

de obicei setarea parametrului presupune specificarea înaintea metodei `accept()` a valorii de timeout). Alți parametri care este necesar să fie setați 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.

În acest model se stabilește o conexiune TCP între o aplicație client și o aplicație server care furnizează un anumit serviciu. Avantajul protocolului TCP/IP este că asigură realizarea unei comunicări stabile, permanente în rețea, existând siguranța că informațiile trimise de un proces vor fi recepționate corect și complet la destinație sau va fi semnalată o excepție în caz contrar.

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

La nivelul serverului, acesta trebuie să creeze întâi un obiect de tip `ServerSocket`. Acest tip de socket nu asigură comunicarea efectivă cu clienții ci este responsabil cu "ascultarea" rețelei și 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 așteaptă "ascultă" rețeaua este `accept()`. Aceasta blochează procesul părinte până la apariția unei cereri și returnează un nou obiect de tip `Socket` ce va asigura comunicarea cu clientul.

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

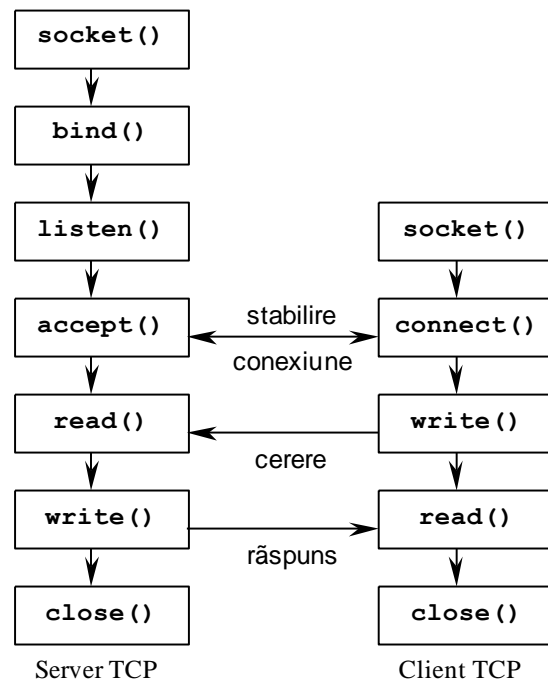


Figura 6.1. Mecanismul socket TCP

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

Fluxurile obținute vor fi folosite împreună cu fluxuri de procesare care să asigure o comunicare facilă între cele două procese. În funcție de specificul aplicației acestea pot fi perechile:

- `BufferedReader`, `BufferedWriter`, și `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. Așteaptă 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 și primește cererea
 - 3.2 Deschide un flux de ieșire și trimite răspunsul
 - 3.3 Închide fluxurile și socketul nou creat

Este recomandat ca tratarea cererilor să se realizeze în fire de execuție separate, pentru ca metoda accept să poată 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 primește răspunsul;
 - 3.3 Inchide fluxurile și socketul creat;

6.2.3. Socketuri datagrame

În 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 recepționează, 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 mașină 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 menționat, 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 verifică ajungerea la destinație a tuturor pachetelor în cazul în care unul nu a ajuns, acesta va fi retrimis automat.

În pachetul Java.net există clasele *DatagramSocket*, *DatagramPacket* și *MulticastSocket* care oferă suport pentru programarea transmisiei de date folosind datagrame. Fiecare datagramă conține un antet (header) și o zonă de date. Antetul cuprinde informațiile de adresare (portul și adresa sursă și destinație) și alte informații legate de asigurarea transmisiei. Datagramele au lungime fixă de aceea este nevoie uneori să fie împărțite în pachete și refăcute la destinație. Transmisia de datagrame se realizează fără stabilirea prealabilă a unei conexiuni, pachetele transmise nu trebuie recepționate în aceeași ordine și deasemenea un socket de tip datagramă poate recepționa 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 citați în buffer.
- *DatagramPacket (byte buffer [], int length, InetAddress address, int port)* – pachetul va fi livrat la adresa de host și portul specificate. Este necesar ca la celălalt capăt al comunicației să existe un server UDP care să asculte conexiuni. Volumul de date care poate fi plasat într-un pachet depinde de tipul protocolului, și de tipul tehnologiei utilizate.

Metodele acestei clase sunt:

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

Clasa DatagramSocket

Această clasă permite transmitia și recepția 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 și legat la o adresă locală.

Transmisia și recepția datagramelor

După crearea unui DatagramPacket și construirea unui DatagramSocket pachetul acesta va fi transferat folosind metoda send(), respectiv recepționat 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.

În cazul în care re folosim pachete, putem schimba conținutul acestora cu metoda setData, precum și adresa la care le trimitem prin setAddress, setPort și 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: getAddress, getPort și getSocketAddress.

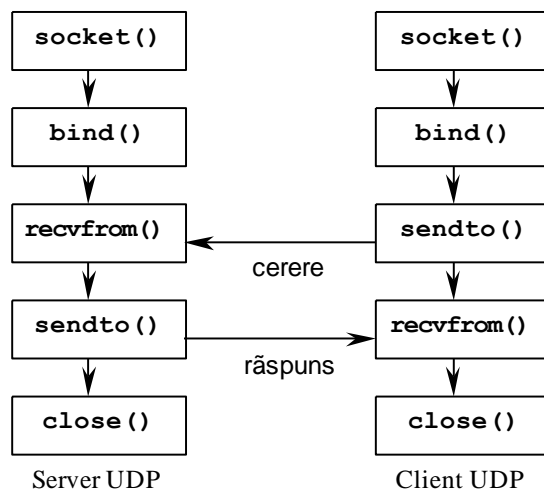


Figura 6.2. Mecanismul socket datagrama

6.2.4.Socketuri multicast

Transmisia (recepția) multicast oferă un set de avantaje de loc de neglijat între care simplificarea proiectării și reducerea consumului de lățime de bandă pentru un set de aplicații 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 recepționa 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 și recepționa date. Asocierea la grup anunță ruterele aflate în calea către serverul care oferă serviciul să transmită datele în mod multicast, hostul fiind responsabil cu adresarea pachetelor în mod multicast. Atașarea la grup determină comunicația similară celei utilizate la transferul de datagrama, trebuie remarcat faptul că doar pentru a transmite în mod multicast nu este necesară atașarea la un grup multicast.

Ceea ce diferențiază totuși comunicarea multicast de cea unicast o constituie specificarea și gestionarea câmpului TTL (Time to live) câmp ce permite evitarea buclor 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* () utilizați în procesarea datagramelor multicast sunt:
MulticastSocket() throws *SocketException* – crează un socket legat la portul anonim (orice port asignat de sistem) și util pentru clienții 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 interfeței utilizate pentru transmisia multicast în hosturile multihomed (cu mai multe adrese IP).

Pentru recepția 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 transmiterea 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.Proiectați o aplicație care să implementeze comunicația între un client și un server respectând specificațiile: programul server ascultă rețeaua și așteaptă să primească o cerere de la un client. Dacă cererea este recepționată fără conflicte, serverul procesează cererea și închide conexiunea. Clientul folosește o adresă pentru a solicita o conexiune unui calculator în rețea (în mod uzual solicită un serviciu unui server), apoi se deconectează din rețea.

6.5.2. Propuneți o soluție de server neblocaant care să utilizeze valori de timeout. Proiectați un server concurent care să genereze pentru fiecare conexiune acceptată un nou fir de execuție.

6.5.3. Să se implementeze o aplicație de tip client-server în care pentru a accesa serverul, acesta solicită o parolă pentru autentificarea unui client apoi va trimite în rețea un fișier către programul client. Programul client se va conecta la server, va trimite parola solicitată, va solicita un fișier și 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 execuția unei aplicații distribuite pe mai multe mașini virtuale Java, permițând ca un set de calculatoare să poată colabora prin transfer de date și cod pentru rezolvarea unui task comun. Tehnologia RMI furnizează o platformă solidă pentru aplicații bazate pe obiect distribuite, fiind utilizabilă atât pentru a conecta obiecte Java între ele, cât și cu obiecte implementate în alte limbaje, oferind posibilitatea extinderii sistemului distribuit într-un mod incremental, prin adăugarea de noi clienți și/sau servere Java conform cerințelor .

7.1.Obiective

- Studiul conceptelor și 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 și 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 facilități specifice programării distribuite obiectuale, cele mai importante fiind:un micro-sistem de numire a entităților, colectorul de deșeuri, 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 aplicațiile distribuite atât datele cât și codul (executabil pe procesoare multiple în sistem), respectiv utilizatorii care comunică și interacționează pot fi distribuiți. RMI este un sistem de programare obiectual, distribuit construit în platforma Java, facilitate ce permite interacțiuni între obiecte care rulează pe mașini virtuale Java (JVM) distribuite în rețea. Spre deosebire de programarea bazată pe socketuri, mecanismul RMI oferă un context de programare distribuită flexibil și relativ simplu, reprezentând mecanismul suport pentru diverse alte modele de dezvoltare a aplicațiilor distribuite Java (de e. Java EE Beans).

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

Sistemul de programare a aplicațiilor distribuite RMI oferă dezvoltatorului de aplicații următoarele facilități:

- *localizarea obiectelor la distanță*- aplicațiile folosesc diverse mecanisme pentru a obține referințele obiectelor remote (serviciul de nume rmiregistry sau transferul referințelor la obiecte ca parte a operației solicitate)
- *comunicare cu obiectele distribuite* - detaliile comunicării sunt gestionate de RMI, pentru programator comunicația remote este similară 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 și 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 și apoi va invoca metoda. Pot fi utilizate servere Web pentru a încărca codul intermediar al claselor Java client-server pentru obiectele aplicației, 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 și ca valori returnate.
- *oferă cod portabil*: raportat la JVM
- *integrează un Garbage Collector distribuit* oferind astfel suport pentru gestionarea referințelor la obiectele distribuite
- *integrează mecanisme de securitate*: folosește controlerul de securitate integrat JVM pentru a proteja sistemele de cod destructiv
- oferă suport pentru procesare paralelă și comportament mobil: poate să schimbe ‘comportamentul’, adică implementările claselor, de la client la server și invers

7.2.2. Arhitectura RMI

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

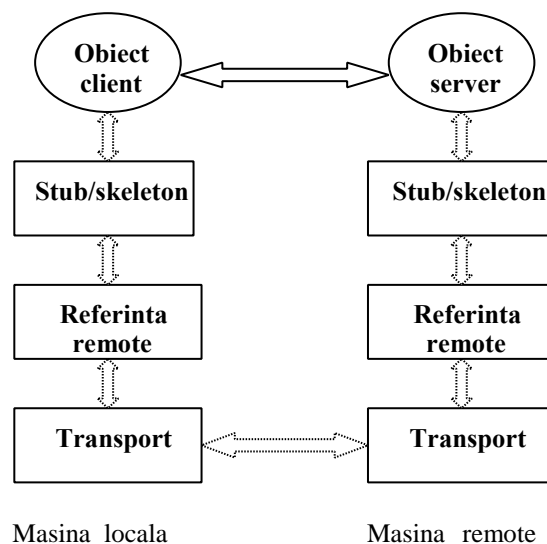


Figura 7.1.Arhitectura RMI și relația client-server

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 distanță parcurge nivelurile sistemului RMI în ambele sensuri, presupunând folosirea unui stub pentru obiectul de la distanță și a unei căi spre acel obiect. Acest nivel definește interfața dintre aplicație și sistemul RMI, nu se ocupă cu detalii de transport, dar transmite datele spre stratul de referință la distanță via abstractizarea numită *stream-uri*, bazat pe mecanismul de serializare, ce susține transmisia obiectelor Java între diferite spații de adrese. Clasele de tip stub -skeleton sunt generate la runtime și sunt încărcate dinamic.

Stubul pentru un obiect remote este proxy-ul la client al acelui obiect, el implementează toate interfețele care sunt suportate de obiectul remote, fiind responsabil pentru:

- inițierea unui apel spre obiectul remote (prin apelul stratului de referință la distanță)
- structurarea argumentelor (argumente sunt obținute de la stratul de referință la distanță)
- informarea stratului de referință la distanță de faptul că apelul este necesar
- reconversia valorii de return sau a excepției dintr-un stream
- informarea stratului de referință la distanță de faptul că apelul a fost efectuat

Skeletonul pentru un obiect remote este o entitate în server care conține 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 excepției în stream-ul marshall

Nivelul de referință la distanță integrează comportamentul referinței la distanță a obiectului, se ocupă cu interfața de nivel jos a nivelului de transport și cu gestionarea unui protocol specific de referință la distanță independent de stub-rile clienților și skeleton-ii serverelor. Nivelul de referință la distanță are două componente care cooperează astfel: componenta client conține informații specifice despre server-ul remote și comunică via strat transport cu componenta server. Fiecare implementare de obiect remote își alege propria sa subclasă de referință 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 referință persistentă spre obiectul remote
- strategii de reconectare

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

- realizarea conexiunii spre spațiul de adrese îndepărtat
- managementul conexiunilor, monitorizarea stării conexiunii
- așteptarea de apeluri și realizarea unei conexiuni pentru un apel cerut de client
- păstrarea unei tabele de obiecte remote care există în spațiul de adrese
- localizarea dispecerului pentru destinația unui apel la distanță și transmiterea conexiunii
-

Reprezentarea concretă a referinței unui obiect remote constă dintr-un capăt de conexiune și un identificator de obiect, reprezentare numită "referință vie" (live reference). Fiind dată o asemenea reprezentare pentru un obiect remote, un transportor poate folosi capătul de conexiune pentru realizarea unei conexiuni la spațiul de adrese în care se găsește obiectul remote. Transportorul conține 4 abstracții de bază:

1. *capăt de conexiune* - abstractizarea unui spațiu de adrese sau a unei JVM, pentru un capăt de conexiune se poate obține o instanță specifică pentru transport
2. *canal* - abstractizarea pentru calea dintre două spații de adrese, responsabil pentru managementul conexiunilor între spații de adrese locale și spațiul de adrese remote pentru care este definit canalul
3. *conexiune* - abstractizare pentru transferul datelor

4. *abstracția de transport* controlează canalele. Fiecare canal e o conexiune virtuală între 2 spații de adrese, într-un transport poate exista un singur canal pentru fiecare pereche de spații de adrese, abstracția de transport fiind responsabilă pentru acceptarea apelurilor de pe conexiunile care se stabilesc spre spațiul de adrese și pentru pasarea acestui apel spre straturile superioare din sistem pentru același transport pot exista mai multe implementări (TCP-UDP pentru aceeași mașina virtuală).

7.2.3. Mecanisme RMI

Modelul obiectelor distribuite. Numim obiect remote un obiect care oferă metode apelabile de la distanță. 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 distanță este realizată prin intermediul unui *server de nume (rmiregister)*, care stochează *referințe tip URL către obiecte la distanță* (protocolul din URL poartă numele rmi). Un astfel de serviciu de nume necesită următorul set de funcțiuni: introducere de asocieri nume – obiect, căutări de asocieri, modificări / ștergeri de asocieri, listarea asocierilor existente la un moment dat.

Serializare. Mecanismul prin care un obiect este transformat într-un flux de octeți și mai apoi reconstituit, se numește serializare, fiind deosebit de util pentru RMI și alte aplicații în rețea.

Pentru transmiterea efectivă la distanță a parametrilor și a rezultatelor, RMI folosește *mecanismul de serializare* (Object Serialization Protocol), protocol ce se bazează pe mecanismul obișnuit de serializare. Stub-ul și skeleton-ul se ocupă cu detaliile de rețea (împachetarea și despachetarea parametrilor și a valorilor returnate de către metode apelabile la distanță) transformând parametri și valorile returnate în fluxuri de octeți ce pot fi transmise prin rețea.

În Java, orice obiect care implementează interfața *java.io.Serializable* poate fi transformat într-un flux de octeți și mai apoi reconstituit din acesta. Această interfață nu adaugă nici o metodă necesar a fi implementată, ea este o interfață ce poate fi utilizată pentru a permite serializarea claselor, în acest mod pot fi transferate obiecte complexe Java în rețea, 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 distanță trebuie să fie serializabil. Cerințe de serializare a unei clase sunt:

- să posede un constructor public fără parametri, necesar pentru deserializarea corectă a obiectelor
- să nu conțină referințe 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).

Interfața la distanță. Legătura între server și client este realizată prin intermediul unei interfețe de acces la distanță. Definierea unei astfel de interfețe trebuie să verifice următoarele condiții:

- trebuie declarată public
- trebuie să extindă interfața *java.rmi.Remote*
- fiecare metodă din interfața la distanță trebuie să declare posibilitatea de a arunca excepția *java.rmi.RemoteException*

Implementarea interfeței la distanță respectă *următoarele etape*:

1. obiectul server extinde *java.rmi.server.UnicastRemoteObject* și implementează interfața la distanță
2. se specifică constructorul obiectului server (și excepția *java.rmi.RemoteException*)
3. pentru obiectul server trebuie să se definească și să se instaleze un manager de securitate (de ex. *RMISecurityManager*).
4. constructorul obiectului server folosește, printre altele, un fișier în care să se specifice “politica de securitate” adoptată de către server (identificarea numelui acestui fișier 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 (rmiregistry) sub numele dorit (java. rmi. Naming. bind() sau java. rmi. Naming. rebind()).
6. se implementează metodele apelabile de la distanță (interfețe la distanță).

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

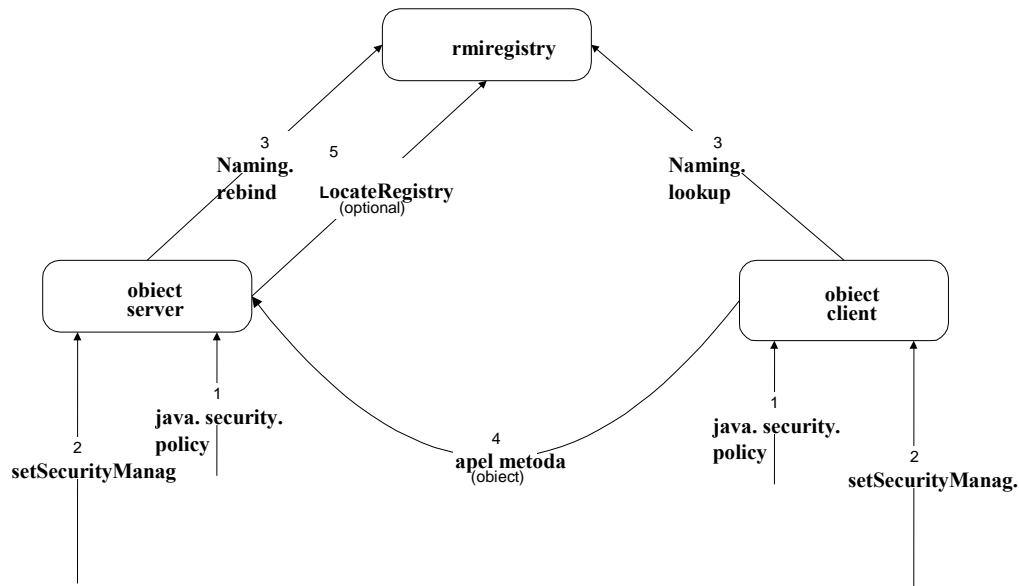


Figura 7.2.Scenariul utilizării RMI

7.2.4. Clase și interfețe RMI

Clase și interfețe client. În dezvoltarea părții de client a unei aplicații se utilizează interfața *Remote* și clasele *RemoteException* și *Naming*, alături de o serie de excepții derivate din *RemoteException* și clasa *RMISecurityManager* utilizată la încărcarea claselor stub în aplicații.

Clasa Naming constituie mecanismul prin care clienții pot obține referiri la obiectele la distanță (ea este folosită și de server pentru înregistrarea obiectelor cu rol de server la distanță), localizarea obiectelor la distanță are la bază mecanismul URL, clasa are doar metode statice și nu definește constructori, metodele sale uzuale sunt *lookup ()*, *String()*, *bind()*, *rebind()* *unbind()* utilizate pentru înregistrarea serverelor.

Clase și interfețe server. Exceptând clasa *RMISecurityManager* care se găsește în pachetul *java. rmi*, toate celelalte clase și interfețe utilizate în programele server se găsesc în pachetul *java.rmi. server*.

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

Clasa RemoteServer este o superclasă pentru implementări de servere și oferă o gamă variată de semantici pentru referirile la distanță. Toate metodele clasei sunt statice. Metoda *getClientHost* apelată dintr-un fir de execuție aflat în cursul tratării unui apel de metodă la distanță 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 clasă derivă din RemoteServer și 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 viață 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 absența managerului de securitate, RMI nu va putea încărca clasele stub decât din sistemul local de fișiere (CLASSPATH). Managerul de securitate permite aplicației să determine înaintea oricărei operații potențial periculoase dacă ea se va efectua prin încărcătorul de clase sau din sistemul local, astfel aplicația poate autoriza sau dezautoriza operația. Dacă operația nu e permisă se va genera excepția 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 încărcătoare și nici să execute metodele clasei ClassLoader
- *checkPropertiesAccess() (String key)* – stuburile nu pot accesa lista de proprietăți 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 aplicații pentru a încărca clase de la un URL. Metodele sale sunt:

- *public static class loadClass(String name)* – folosește proprietatea java. rmi. server. codebase pentru a seta 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 execuție RMI localizarea claselor stuburi și parametri/valori returnate ce vor fi disponibile clienților prin această proprietate.

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

Clasa abstractă RMI SocketFactory – specifică modul în care transportul RMI trebuie să obțină 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** conține două interfețe Registry și RegistryHandler precum și clasa LocateRegistry ce realizează serviciul de înregistrare și regăsire a obiectelor la distanță folosind nume simple. E permis ca fiecare process server să își definească propriul registru de obiecte sau să folosească un singur registru pentru un host. Registrul este un obiect la distanță la rândul său. Interfața registry specifică metodele de *căutare, legare, relegare, ștergere și listare* a conținutului unui registru. Operații bind/rebind/unbind sunt premise doar dacă solicitanții sunt pe același host ca și serverul însă operația lookup este permisă de oriunde.

Clasa LocateRegistry - conține metodele statice ce returnează o referință la un registru, o metodă de creare registru, referințele returnate sunt referințe la un stub la distanță 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 și pentru portul specificat (portul pentru Registry este 1099). Metoda *CreateRegistry* creează și exportă un registru pe hostul local, registru ce implementează un serviciu de nume simplu, în care numele unui obiect la distanță (String) este asociat unei referințe la distanță. Legăturile astfel create sunt valabile doar pentru activarea curentă a registrului. Interfața *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 și interfețe pentru stuburi și skeletoane

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

```

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 abstracțiunea utilizată de stuburi și scheletoane pentru a transfera un apel la un obiect remote și 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 ieșire, după ce marchează în antet informație referitoare la succesul apelului, obținerea fluxului cu rezultate trebuie să reușească o singură dată pentru fiecare apel. Metoda executeCall conține ceea ce este necesar pentru efectuarea apelului, iar done eliberează resursele după terminarea apelului remote. Versiunile post JDK 1. 4. au depreciat această interfață în favoarea utilizării metodei invoke.

Interfața **RemoteRef** – definește un descriptor (handle) pentru un obiect la distanță, fiecare stub integrează câte un exemplar RemoteRef ce conține reprezentarea concretă a referinței. Metoda NewCall permite crearea unui obiect de apel adecvat pentru un nou apel de metodă la distanță obj. Tabloul de operații op conține operațiile disponibile asupra obiectului la distanță iar num este un indice în acest tablou prin care este indicată operația 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(); }

```

Interfața **ServerRef** reprezintă descriptorul de server pentru implementarea unui obiect la distanță. Metoda exportObject caută sau creează un obiect stub client pentru implementarea obiectului Remote obj. Astfel server este obiectul server la distanță pentru implementare poate fi identic cu obj), iar data conține informațiile 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; }

```

Interfața **Skeleton** este utilizată în implementarea skeletoanelor generate de compilatorul rmic. Metoda `dispatch` deordonanțează argumentele din fluxul de intrare obținut de la obiectul `call`, apelează metoda `opnum` pe implementarea efectivă a obiectului `obj` și 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 - menține descrierea unei metode pentru un obiect la distanță
public class Operation{ public Operation (String op);
public String getOperation(); public String toString(); }
```

Securitatea RMI

RMI pune la dispoziție pentru securizarea canalelor de comunicație între client și server un mecanism de izolare a implementărilor aduse într-un ‘sand box’ securizat pentru a proteja sistemul de posibile atacuri efectuate de clienți nesiguri. Este important a se defini nevoile de securitate pentru a stabili reguli stricte ce se vor aplica pentru toți clienții. S-ar putea să fie nevoie a se securiza un canal de comunicație între client și 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 comunicație între client și server. Începând cu JDK 1. 2 se vor putea specifica cerințele pentru serviciile oferite pentru socket-urile serverului. 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 acțiune ce implică un nivel de securitate mai deosebit, cum ar fi deschiderea de fișiere, stabilirea conexiunilor pentru rețea etc. RMI furnizează un tip *RMI SecurityManager*, 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 și instala un obiect propriu de securitate pentru a forța diferite politici de securitate specifice.

Nu este însă necesar și nici recomandabil să se altereze politica implicită a JDK, ci este preferabil să se definească o politică particulară aplicației ce ridică probleme. Pentru a lansa în execuție o aplicație Java cu altă politică decât cea implicită se poate folosi un fișier de configurare care să definească noua politică. Apoi, mașina virtuală trebuie instruită să folosească noua politică plasând numele fișierului respectiv în proprietatea globală *java. security. policy*.

Un asemenea fișier ar putea avea următorul conținut:

```
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 și rezolvarea claselor în RMI. Adnotarea claselor denotă mecanismul prin care în fluxul de date serializat *se adaugă și o adresă de unde se poate obține 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 instanțiat direct din *ObjectOutputStream* se utilizează un obiect instanțiat din clasa derivată *MarshalOutputStream*, în care se definește metoda *annotateClass()*. Pentru a încărca clasele adnotate

(operația inversă serializării) se utilizează un obiect instanțiat din `MarshalInputStream` care derivă din `ObjectInputStream` și care definește 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 distanță*. În acest caz (de un `URLClassLoader` spre exemplu), se interoghează încărcătorul respectiv asupra adresei de origine și 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 proprietății `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 recepțează trebuie să determine prin alte mijloace adresa de la care le poate încărca.

Rezolvarea claselor în RMI. «Rezolvarea» unei clase înseamnă obținerea 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 instanțierea obiectului.

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

- pe baza adnotării aferente clasei transmisă prin RMI Wire Protocol
- pe baza unei proprietăți globale pentru toate clasele recepționate 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 și la recepționarea fluxului serializat, mecanism numit *adnotare implicită*. Încărcătorul special menționat poartă denumirea de încărcător contextual (context class loader). Acest mecanism este specific Java 2 și este foarte flexibil deoarece pentru fiecare fir de execuție se poate specifica un încărcător contextual diferit. La instanțierea unui nou fir de execuție, acesta moștenește 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 forțat chiar dacă clasa este adnotată în fluxul serializat, cu condiția ca proprietatea globală `java.rmi.server.useCodebaseOnly` să aibă valoarea `true`), însă pentru ca adnotarea să fie luată în considerație 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 și apoi încărcătorul contextual. Evident, în absența unui manager de securitate se va încerca direct încărcătorul contextual, iar dacă acesta eșuează se va afișa 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 și de cel care le recepționează, însă este necesară stabilirea unei politici adecvate în acest sens.

Să presupunem că dorim să realizăm o aplicație, în care un server transmite un comportament care va fi executat de către clienți. Există trei posibilități și anume:

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

- setarea proprietății codebase și la client și 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 funcționa indiferent unde se specifică proprietatea codebase.

Să presupunem însă că dorim să ne conectăm cu clientul la mai multe servere simultan și că nu toate serverele au codul comportamentului la aceeași 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 utilă a doua soluție, în care fiecare server va adnota corespunzător fluxul său de date, astfel încât clientul să poată obține fiecare comportament de la sursa corespunzătoare.

b) Flux de date, transfer de comportament

O aplicație care dorește să transfere comportament va utiliza două fluxuri de informație: fluxul de date (transmis prin RMI Wire Protocol) și fluxul de cod binar Java.

Fluxul de date: RMI Wire Protocol. Funcționarea RMI se bazează pe protocolul RMI Transport Protocol care realizează implementarea fluxului de date între apelant și apelat (protocol de nivel aplicație), 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 și încapsulate, în mod implicit atât conexiunile directe cât și cele multiplexate se efectuează peste TCP.

Conexiunile directe pot fi utilizate atunci când nu există restricții de securitate la nivelul mașinii virtuale a serverului și nici la nivelul rețelei. Conexiunile multiplexate sunt utile atunci când se dorește exportarea unor obiecte dintr-un applet (exportarea este acțiunea prin care un obiect devine invocabil de la distanță).

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 reuși, iar obiectul va fi apelat direct. În caz contrar, se trimite cererea către portul 80 al serverului, în speranța că firewall-ul permite conexiuni către porturi HTTP standard.
- Acces indirect – e necesar ca pe mașina pe care se găsește 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 și 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 distanță și obținute 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 fișiere suportate de mașina 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 folosește un URLClassLoader pentru a transfera codul binar al claselor ce prezintă adnotare.

c) Redefinirea socketurilor RMI

O facilități 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 capacități speciale pentru situațiile în care *se impune o anumită politică de securitate în rețea* și trebuie să existe control asupra porturilor utilizate de RMI. Aceasta este realizată 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 comunicațiilor prin utilizarea unui mecanism specific numit Custom RMI Socket Factory.

*RMI*SocketFactory face posibilă construirea unei “fabrici” care să producă socketuri, însă odată instanțiată “fabrica” aceasta putea produce numai socketuri de același tip. Prin introducerea clasei *java.rmi.server.SocketType* se oferă posibilitatea construirii unui *RMI*SocketFactory 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 restricții destul de severe au făcut ca această facilități să nu poată fi exploatată corespunzător în JDK 1. 1, astfel:

- aceeași fabrică de socluri era necesar a fi folosită pentru toate obiectele invocabile de la distanță,
- era necesar să existe un registru RMI (*rmiregistry*) pentru fiecare tip de socluri, iar
- clasele fabricii de socluri nu puteau fi aduse de la distanță 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 interfețe:

```
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 inițiază o conexiune, tip server ascultă în așteptarea unei conexiuni) utilizate de un obiect invocabil la distanță se stabilește la momentul exportării obiectului. Presupunând că obiectul este derivat din *UnicastRemoteObject*, va fi suficient să-l construim prin intermediul unui constructor care primește 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 instanția socluri pe care se vor primi conexiuni de la clienți, iar fabrica de socluri client nu e folosită ca atare de obiectul server, ci se va transmite clienților pentru ca aceștia să se poată conecta la server. Putem imagina un client care contactează un server prin RMI « obișnuit », stabilește de comun acord cu acesta ca sesiunea să se desfășoare criptat, după care utilizează o fabrică de socluri cu criptare pentru restul comunicației, iar fabrica ce generează soclurile cu criptare poate fi transmisă de la server la client pe parcursul execuției acestuia.

d)Apeluri tip Callback

În programarea concurentă, noțiunea de invocare la distanță integrează un aspect suplimentar: posibilitatea ca mașina server să « întoarcă un răspuns », adică să determine executarea unei metode pe mașina client, facilități suplimentară ce poartă numele de apel invers (callback). Pentru a putea realiza apelurile inverse este necesar să se definească o nouă interfață la distanță și să se implementeze aceasta interfață de către client, iar interfața la distanță implementată de server va trebui să prevadă o metodă prin care clienții își pot înregistra la server interesul pentru a fi anunțați de apariția anumitor modificări în starea serverului. Principial, lucrurile decurg analog ca la invocarea la distanță a unei metode. Fie *metC* metoda care trebuie invocată pe mașina client din cadrul metodei *metS*. Este evident necesar ca pe mașina client să fie creat un obiect *ObC* de tipul clasei care include metoda *metC*, iar mașina server să primească o referință la acest obiect, pentru ca (pe baza acestei referințe) să poată invoca metoda *metC*, ce va fi executată pe mașina client. Implementarea apelurilor inverse va fi mai simplă, deoarece legătura între mașini este deja stabilită.

Pentru realizarea apelurilor inverse, trebuie întreprinse doar următoarele două acțiuni: 1) crearea obiectului și pregătirea lui pentru a fi exportat; 2) transmiterea obiectului mașinii server. Prima acțiune presupune ca *ObC* este o instanță a unei clase ce extinde interfața *Remote*. Pregătirea sa pentru

export este realizată prin invocarea următoarei metode publice și statice a clasei `UnicastRemoteObject`: `RemoteStub exportObject(Remote ObC)` throws `RemoteException`. A doua acțiune poate fi realizată simplu, comunicând serverului obiectul `ObC` prin includerea lui ca argument la invocarea unei metode prin intermediul referinței la obiectul `Obs`. Se presupune că obiectul `ObC` este de tipul unei clase serializabile.

e) Obiecte activabile

Mecanismul RMI se îmbogățește cu obiecte activabile la distanță. Acestea reflectă un mecanism prin intermediul căruia se pot activa obiecte server *doar atunci când este nevoie de ele și se folosesc identificatori persistenti pentru a putea referi obiectele și atunci când acestea sunt inactive*, aceasta conduce evident la o folosire mai eficientă a memoriei pe mașina server.

De lansarea în execuție a acestor obiecte activabile este responsabil un demon (`rmid`) ce lansează mașini virtuale Java în care rulează efectiv obiectele. Pentru a nu crea un număr excesiv de mașini virtuale și pentru a permite cooperarea între obiectele server, acestea pot fi adunate în grupuri de activare; toate obiectele din același grup fiind executate în aceeași mașină virtuală. Prin introducerea clasei `java.RMI.activation.Activable` și a demonului RMI, `rmid`, obiectele remote pot fi create și executate atunci când este nevoie de ele, programele trebuie doar să înregistreze informațiile de implementare despre obiectele remote și demonul `rmid` va furniza instanța 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 evolute 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 interfață prin care poate să trimită mesaje spre camera de discuții. Un mesaj trimis de un utilizator va apărea simultan în 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 dispoziție un arbore de directoare și fișiere din care poate trimite, la cerere, fișiere către clienți. Clientul se conectează la server și solicită fișiere. 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 fișierul, `dir` - cere lista de fișiere de la server, `cd` - schimbă 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) inițiază comunicarea, conectându-se la adresa celeilalte (B) și specificând un nume de utilizator cu care dorește 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 același 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 fișiere care conțin tot timpul ultima versiune a unui sistem software. Mai multe alte calculatoare doresc să aibă imagini oglindă ale distribuției (mirrors) pe care să le mențină cât mai la zi. Pentru aceasta, pe server există un program server care gestionează distribuția, iar pe fiecare calculator "mirror" câte 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 fișierelor, 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 trimisă către client ca și răspuns. La primirea comenzii exit termină conexiunea. Clientul se conectează la server și trimite comenzi către acesta. Primește rezultatul și 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 execuție 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-plugin-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 și coordoneze acțiunile, motiv pentru care la nivelul unui sistem distribuit apare o coordonată suplimentară și anume timpul. Sincronizarea și coordonarea joacă un rol foarte important în majoritatea sistemelor distribuite. Prin *coordonare* se face referire la corelarea acțiunilor proceselor separate și asigurarea unor mecanisme de comunicare la nivel global. *Sincronizarea* este similară coordonării însă integrează și factorul timp, referind ordinea evenimentelor și execuția instrucțiunilor raportată la timp. Un exemplu al sincronizării este ordonarea evenimentelor distribuite dintr-un fișier de tip log file și certitudinea că un proces realizează o anumită acțiune 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 și implementarea unor algoritmi de coordonare, sincronizare a proceselor și 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 același *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 (așa 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 relevanță într-un sistem distribuit. În primul rând, un ceas intern ține evidențe 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, același pentru toate componentele, și 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 și 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 distribuiți, în general, sunt utilizate două tipuri de modele de timp și anume un model sincron, timpul necesar realizării tuturor sarcinilor, întârzierile din timpul comunicării și diferențele 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 țin evidența timpului fizic. În sistemele distribuite care se bazează pe timpul efectiv, este necesară menținerea 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) și Network Time Protocol (NTP) sunt exemple de algoritmi dezvoltăți pentru sincronizarea semnalelor cu o sursă de timp externă și 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 aplicații, *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 și primirea unui mesaj singular).

Fiind dat un sistem ce conține N procese p_i , i luând valori între $\{1, \dots, N\}$, se definește o ordonare locală a evenimentelor (notată \rightarrow) ca o relație 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 relație de tipul *happened before* \rightarrow , definiția dată de Lamport fiind următoarea:

Relația \rightarrow este astfel încât:

1. $e \rightarrow_i e'$ implică $e \rightarrow e'$,
2. pentru fiecare mesaj m , $send(m) \rightarrow receive(m)$, și
3. $e \rightarrow e'$ și $e' \rightarrow e''$ implică $e \rightarrow e''$ (tranzitivitatea).

Relația \rightarrow este asimilată unei relații de ordonare parțială (îi lipsește reflexivitatea). Dacă $a \rightarrow b$, atunci spunem că a îl afectează cauzal pe b . Considerăm că evenimentele neordonate sunt *concurrente*.

Ca un exemplu, considerăm Figura 1. Avem următoarele relații cauzale:

$$\begin{aligned} E_{11} &\rightarrow E_{12}, E_{13}, E_{14}, E_{23}, E_{24}, \dots \\ E_{21} &\rightarrow E_{22}, E_{23}, E_{24}, E_{13}, E_{14}, \dots \end{aligned}$$

Mai mult, următoarele evenimente sunt concurrente $E_{11}||E_{21}$, $E_{12}||E_{22}$, $E_{13}||E_{23}$, $E_{11}||E_{22}$, $E_{13}||E_{24}$, $E_{14}||E_{23}$.

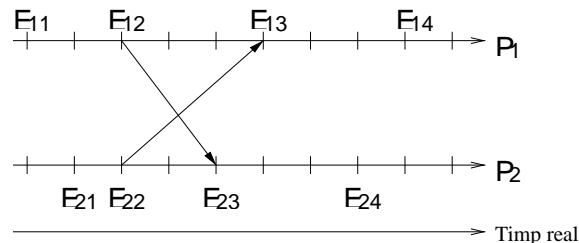


Figura 8.1. Ordonarea evenimentelor

8.2.2. Ceasuri logice Lamport

Ceasurile logice Lamport pot fi implementate sub forma unor numărătoare care calculează local relația de tipul *happened-before* \rightarrow . Aceasta înseamnă că, fiecare proces p_i reține 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 și $L(e)$ denotă un timestamp al evenimentului e la procesul la care acesta a apărut.

Evoluția la nivel de proces este următoarea :

1. Înainte de a atribui un timestamp unui eveniment, 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_j execută $L_j := L_j + 1$ și trimite noul L_j cu m .
 - Procesul p_j recepționează L_i cu m și 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 $E_{12} \rightarrow E_{23}$ și $L_1(E_{12}) < L_2(E_{23})$, totodată avem și $E_{13} \rightarrow E_{24}$ cât timp $L_1(E_{13}) < L_2(E_{24})$.

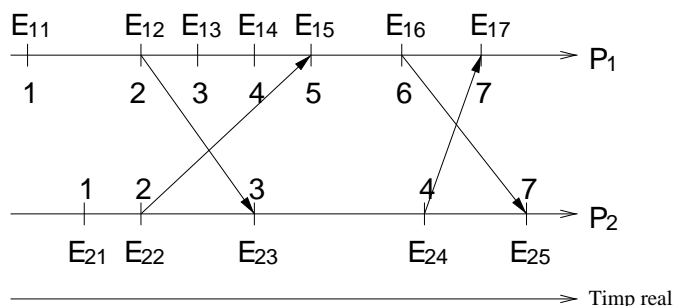


Figura 8.2. Exemplu de utilizare a ceasurilor Lamport

În unele situații, de exemplu, implementarea lacătelor distribuite pentru algoritmi de excludere mutual distribuită, o ordonare parțială a evenimentelor nu este suficientă și deci, este necesară o ordonare totală. În aceste cazuri, ordonarea parțială poate fi completată pentru obținerea unei ordonări totale, prin includerea identificatorilor de procese. Dându-se următoarele stampe de timp locale $L_i(e)$ și $L_j(e')$, definim *stampe de timp globale*, apoi utilizăm ordonarea standard lexicografică.

8.2.3. Ceasuri logice vectoriale

Principala deficiență a ceasurilor logice Lamport este faptul că $L(a) < L(b)$ nu implică $a \rightarrow b$; prin urmare, nu putem deduce dependențe cauzale din timestamps. De exemplu, în Figura 3, avem $L_1(E_{11}) < L_3(E_{33})$, dar $E_{11} \not\rightarrow E_{33}$.

Esența problemei este reprezentată de înaintarea independentă sau prin mesaje a ceasurilor, fără menținerea unui istoric al acestei înaintări. Problema poate fi rezolvată prin trecerea de la ceasuri scalare la vectorii de ceasuri, unde fiecare proces reține un vector V_i .

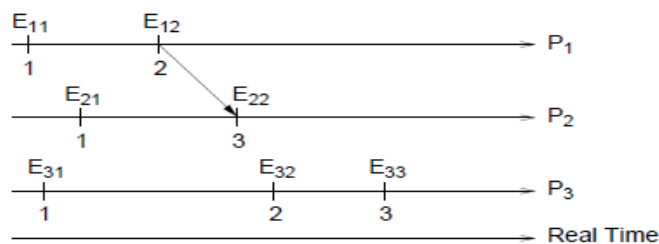


Figura 8.3. Exemplu al lipsei de cauzalitate cu ceasuri Lamport

V_i este un vector de mărime N , unde N este numărul de procese. Componenta $V_i[j]$ conține informațiile pe care le deține procesul p_i referitoare la semnalul p_j . Inițial, avem $V_i[j] := 0$ pentru $i, j \in \{1, \dots, N\}$. 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 $V_i[i] := V_i[i] + 1$ și trimite V_i cu m .
 Procesul p_j primește V_i cu m și adună semnalele de vector V_i și V_j astfel:

$$V_j[k] := \max(V_j[k], V_i[k]) + 1, \text{ dacă } j = k$$

$$V_j[k] := \max(V_j[k], V_i[k]), \text{ dacă } j \neq k$$

Astfel se asigură faptul că tot ceea ce se întâmplă ulterior la procesul p_j , este acum în relație 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 și numai dacă $V(a) < V(b)$. De exemplu, considerăm notațiile din diagrama reprezentată în Figura 4. Fiecare eveniment este notat atât cu valoarea vectorului de ceas propriu (triplu), cât și valoarea corespondentă a unui ceas scalar Lamport.

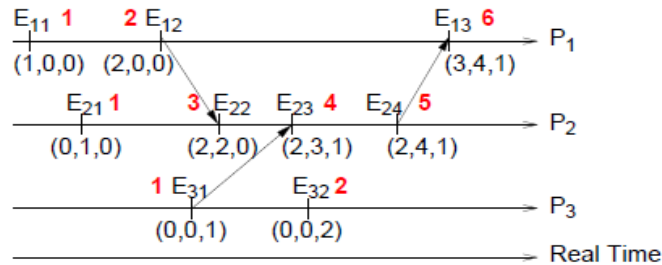


Figura 8.4. Exemplu ceasuri vectoriale

8.2.4. Algoritmi pentru ceasuri logice

Pentru cei doi algoritmi, este valid un set de ipoteze generale, și 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 proces cunoaște procesele cărora le trimite mesaje și de la care poate primi mesaje.
- 3: transferul de mesaje poate fi sincron. Astfel este cunoscută o limită superioară a timpului în care pot fi trimise mesaje, inclusiv timpul necesar generării mesajului și 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

Algoritm 1. Birman-Schiper-Stephenson

Scop: menține ordinea la trimiterea mesajelor. De exemplu, dacă $send(m_1) \rightarrow send(m_2)$, atunci pentru toate procesele ce primesc m_1 și m_2 , este adevărată relația $receive(m_1) \rightarrow receive(m_2)$. Astfel, m_2 nu va fi trimis procesului receptor decât după mesajul m_1 , fiind astfel necesar un bufer pentru mesajele aflate în așteptare

- 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 $P_j, j \neq i$, recepționează m cu timestampul t^m , va întârzia livrarea mesajului până când ambele condiții sunt îndeplinite:
 - a. $C_j[i] = t^m[i] - 1$; și
 - b. pentru $k \leq n$ și $k \neq i$, $C_j[k] \leq t^m[k]$.
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-Eggl-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 corect 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 conține 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 stampana 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. P_i trimite message m , timestamp t^m , și V_i , procesului P_j
2. P_i setează $V_i[j] = t^m$

P_j recepționează un mesaj de la P_i

1. Când $P_j, j \neq i$, primește m , va întârzia livrarea 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 $k' = 1, \dots, n$
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 operația de găsim a unui proces neșuat, ce deține 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 proprietăți:

1. Siguranță: un proces, fie nu cunoaște coordonatorul, fie cunoaște identificatorul procesului cu cel mai mare număr de identificare.
2. Longevitate: în cele din urmă, un proces este supus eșecului sau îl cunoaște pe coordonator.

Unul din cei mai cunoscuți algoritmi de alegere lider a fost propus de Garcia-Molina și utilizează trei tipuri de mesaje:

1. Mesaje de tipul Election (alegere) – anunță alegerea
2. Mesaje de tipul Answer (răspuns) – procesele răspund la o alegere
3. Mesaje de tipul Leader (coordonator) – coordonatorii aleși se anunță în sistem

Algoritm Bully. Modul de execuție a acestui algoritm este descris în continuare. Un proces începe o alegere în momentul în care realizează, prin intermediul unui timeout, faptul că recentul coordonator a eșuat, sau dacă primește 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 primește niciun mesaj de tipul Answer într-un interval de timp predefinit, procesul care a început alegerea va decide că trebuie să devină coordonator și va trimite un mesaj de tipul Coordinator către toate celelalte procese.

În cazul în care primește un mesaj de tipul Answer, procesul care a inițiat alegerea va aștepta un interval de timp predefinit pentru un mesaj de tipul Coordinator. Un proces care primește un mesaj de tipul Election poate anunța instantaneu că el este coordonatorul dacă știe sigur că deține numărul de identificare cel mai mare. Altfel, va începe și el o alegere subordonată prin trimiterea unui mesaj de tipul Election către procesele care au numerele de identificare mai mari. Algoritm este numit și algoritm bully (forța 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 cunoaște 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 și 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 și fiecare proces cunoaște informații despre structura inelului. Mesajele implicate sunt doar de două tipuri – mesaje de tipul *Election* și de tipul *Lider*. Un proces începe o alegere în momentul în care realizează că precedentul coordonator a eșuat. O nouă alegere pornește prin trimiterea unui mesaj de tipul *Election* la primul vecin de pe inel. Mesajul conține identificatorul procesului, acesta este transmis prin inel, fiecare proces adăugându-și propriul identificator. Alegerea este completă când mesajul ajunge înapoi la procesul inițiator. Bazat pe conținutul mesajului, procesul respectiv determină numărul de identificare maxim și trimite un mesaj de tipul *Lider*, specificând faptul că procesul respectiv este câștigătorul.

Algoritmul 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 concurenței din cadrul sistemelor distribuite sunt similare cu cele întâlnite în studiul sistemelor de operare și a aplicațiilor multithread. În particular, ne referim aici la problemele impuse de condițiile de concurență 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 și monitoarele. În sistemele distribuite, rezolvarea problemelor datorate concurenței devine mult mai complexă, datorită lipsei de resurse comune directe (cum ar fi memoria, regiștrii CPU, etc.), datorită lipsei unui ceas global, a unei stări globale unice și datorită prezenței întârzierilor de comunicare.

Atunci când se dorește accesul concurrent la o resursă dintr-un sistem distribuit, e necesară existența unor mecanisme care ajută la prevenirea condițiilor de concurență în timp ce procesele se află în cadrul unor secțiuni critice. Aceste mecanisme trebuie să asigure existența următoarelor trei proprietăți: a) siguranța: cel mult un proces are dreptul de a executa secțiunea critică la un moment dat. b) longevitatea: la un moment dat, cererile de a intra și de a ieși din secțiunea 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

specializați ce utilizează ceasurile logice pentru a determina ordonarea și a asigura serializarea accesului la resursă (algoritmii Ricart-Agrawala și Maekawa), soluții ce vor fi descrise în continuare.

Ipotezele generale valabile pentru toți acești algoritmi sunt următoarele :nu există memorie partajată, nu este utilizabil un ceas global, fiecare proces are propriul ceas și 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 și ieșirea din secțiunile critice. Procesele trebuie să trimită cereri de intrare și ieșire dintr-o secțiune critică către un server coordonator. Acesta acordă permisiuni de intrare prin trimiterea unui *token* spre procesul care a înaintat cererea. La părăsirea secțiunii critice, token-ul este returnat serverului. Procesele care doresc să intre într-o secțiune critică în timp ce un alt proces deține token-ul, sunt așezate într-o coadă. Când token-ul este returnat, procesul de la începutul cozii primește token-ul și totodată permisiunea de a intra în secțiunea critică. Această schemă este ușor de implementat, dar nu prezintă o scalabilitate bună, datorită existenței unei autorități centrale, mai mult, este vulnerabilă din punctul de vedere al eșuării 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 secțiunea critică, un proces trebuie să aștepte până când token-ul trece prin dreptul său. În acel moment, procesul reține token-ul până în momentul părăsirii secțiunii 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 lățime de bandă și nodurile sau canalele eșuate pot rupe inelul. O altă problemă este reprezentată de pierderea tokenului – fapt cauzat de prezența eșecurilor. În plus, dacă un proces nou se alătură rețelei sau dorește să o părăsească, este nevoie de logică adițională 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/ieșire dintr-o secțiune critică, întârzierea unui proces înainte ca acesta să primească permisiunea de a intra în secțiunea critică, respectiv nivelul de siguranță al algoritmilor (ce probleme posibile întâmpină aceștia în execuție).

Algoritmul centralizat are nevoie de un total de 3 mesaje inter-schimbate la fiecare execuție a unei secțiuni critice (două pentru intrare și unul pentru ieșire). După ce un proces trimite cererea pentru obținerea permisiunii de a intra într-o secțiune critică, acesta trebuie să aștepte pentru un timp minim în care are loc inter-schimbarea a două mesaje (unul pentru deținătorul curent - ca să returneze token-ul coordonatorului, și unul pentru coordonator – ca să trimită token-ul la procesul aflat în așteptare). Cea mai mare problemă pe care o are acest algoritm referă situația în care coordonatorul eșuează (sau devine nedisponibil) și prin urmare, tot algoritmul eșuează.

Pentru algoritmul inel, numărul mesajelor inter-schimbate per intrare și ieșire din secțiunea critică, depinde de cât de des au nevoie procesele să intre în secțiune. Cu cât mai rar doresc procesele să intre, cu atât mai mult timp va dura drumul token-ului prin inel și cu atât mai mari vor fi costurile, în termenii mesajelor inter-schimbate de intrare într-o secțiune critică. Din punctul de vedere al întârzierilor depinzând de locația token-ului, va exista un număr de 0 până la $n-1$ mesaje înainte ca un proces să intre în secțiune. Cele mai mari probleme întâmpinate de acest algoritm sunt pierderea tokenului și ruperea inelului de către un proces eșuat. Este posibilă, desigur, prevenirea acestora, prin oferirea tuturor proceselor de informații referitoare la structura inelului. În acest mod, nodurile eșuate pot fi evitate.

B) Algoritmi de excludere mutuală ce folosesc ceasuri logice

Algoritm 1. Algoritmul Ricart-Agrawala

Ricart și Agrawala [Ricart- Agrawala 1981] au propus un algoritm pentru excluderea mutuală distribuită, care utilizează ceasuri logice. fiecare proces participant p_i reține un ceas Lamport și 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 secțiunii critice
2. *Wanted*: Așteptând să intre în secțiunea critică
3. *Held*: În interiorul secțiunii critice

Dacă un proces dorește să intre în secțiunea critică, realizează operația de multicast a unui mesaj L_i, p_i și așteaptă 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 secțiunea critică.
- Dacă procesul este în starea *Held*, va întârzia răspunsul la oricare cerere până în momentul ieșirii din secțiunea 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:

- Îmbunătățirea 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 păstrează 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
- Dacă $\text{timestamp} > \text{timp local}$, setează $\text{timp local} = \text{timestamp} + 1$

Acest aspect impune o relație „s-a întâmplat înainte” (happens before) între evenimente, cu proprietățile:

- Dacă evenimentA se întâmplă înaintea evenimentB pe o singură mașină, $T(\text{evenimentA}) < T(\text{evenimentB})$
- Dacă evenimentA este expeditorul mesajului și evenimentB este destinatarul aceluiași mesaj, atunci $T(\text{evenimentA}) < T(\text{evenimentB})$

Această ordonare nu e o ordonare totală, dar poate fi transformată într-una dacă 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 și 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 aceeași și ori va elimina mesajul din grup, ori va opri aplicația. Variante mult mai scalabile ale acestui algoritmul cer fiecărui proces individual să contacteze subseturi ale vecinilor în momentul în care se dorește intrarea în secțiunea critică. Din păcate, eșecul oricărei componente -proces poate afecta intrarea oricăror altor procese în secțiunea critică deci poate periclita execuția algoritmului.

Algoritm 2 .Algoritmul Maekawa

Pentru fiecare proces se definește o mulțime ce conține cereri, notată R_i . Mulțimile de cereri trebuie să aibă proprietățile:

- Intersecția lui R_i cu R_j 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 $K = O(\sqrt{N})$. Algoritmul folosește trei tipuri de mesaje: *Request*, *ok*, *End*. Un proces care solicită intrarea într-o zonă critică trebuie să primească OK-uri doar de la membri mulțimii sale de solicitanți.

Informații suplimentare :

https://en.wikipedia.org/wiki/Maekawa's_algorithm

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 each 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 $3 \cdot \sqrt{N}$ mesaje sunt necesare pentru intrarea intr-o zona critica. Problema posibilă este apariție a unui blocaj (deadlock) la nivelul proceselor, o posibilă rezolvare ar fi următoarea :

- Dacă 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, presuratat CS), sa fie nevoie de $3 \cdot (N-1)$ 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 $2 \cdot (N-1)$, deoarece foloseste mesaje doar de tip request și reply. Mai mult de atât, algoritmul lui Maekawa scade numărul de mesaje la $3 \cdot \sqrt{N}$, 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 algoritmul 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 de date 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 avantaj, dezavantaj, posibile optimizări și număr de mesaje schimbate.

8.4.4. Comparați funcțional algoritmii Ricart-Agrawala și Maekawa, analizând complexitatea, dezavantaj și posibile optimizări.

8.5.Probleme propuse

8.5.1. Implementați algoritmi 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 algoritmi 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 algoritmi 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-subscribe pot fi folosite pentru proiectarea multor modele de comunicat, fiind relative simple eficiente și 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 proiectanților și dezvoltatorilor de a furniza soluții care să reducă sau să elimine limitări în transferul de date între componentele unui sistem informatic, crescând în acest fel productivitatea utilizatorului final și posibilitatea sistemului, în ansamblul lui, de a face față unui volum de date de prelucrat care poate crește 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 aplicațiilor, prin schimbul asincron de mesaje ce oferă fiecărei componente posibilitatea de a prelucra datele în ritmul propriu, fără a afecta performanțele altor componente ale sistemului;
- creșterea capacității sistemului de a face față unor volume mari de date, prin posibilitatea de a adăuga dinamic componente pentru a prelucra concurențial cozi de mesaje omogene;
- posibilitatea proiectării unor sisteme cu arhitecturi flexibile și agile, care se pot ușor 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 rețea. Livrarea asincronă a mesajelor presupune că aplicația care trimite un mesaj nu trebuie să aștepte ca respectivul mesaj să fie recepționat sau prelucrat de aplicația căreia i-a fost destinat; aplicația care produce mesajul are libertatea de a trimite mesajul în cauză și continua apoi propriul flux de prelucrare. Mesajele asincrone sunt manipulate ca unități de sine stătătoare, în sensul în care fiecare mesaj conține toate datele de business și cele de stare necesare pentru a putea fi prelucrat din perspectiva logicii aplicației ce se găsește la destinație. În filosofia de mesagerie asincronă, aplicațiile utilizează o interfață (*Application Programming Interface* - API) simplă pentru a construi mesaje, iar aceste mesaje sunt ulterior pasate către platforma de comunicație orientată pe mesaje, pentru ca aceasta din urmă să le livreze către destinațiile 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 mașinile clienților. De asemenea, protocoalele de comunicație utilizate la nivelul de rețea transport includ TCP/IP, HTTP, SSL și IP multicasting.

Este important de menționat faptul că, în contextul utilizării unei platforme de comunicație orientate pe mesaje, termenul de client capătă o semnificație mai largă, în sensul în care toate aplicațiile conectate la platforma de comunicație devin clienți ai acestui *middleware*, chiar dacă din perspectiva logicii funcționale a sistemului, privit în ansamblul său, pot juca rol de client, respectiv de server. Sistemele de comunicație orientate pe mesaje sunt compuse din mesajele aplicațiilor client ale platformei și componenta de server de *middleware* pentru rutarea acestor mesaje. Clienții transmit mesaje către server de mesagerie, mesaje care sunt apoi distribuite de acesta către alți clienți ai sistemului. Sistemele de mesagerie pot fi centralizate, descentralizate sau hibride.

Sistemele de mesagerie care se bazează pe o *arhitectură centralizată* folosesc un așa-numit server de mesaje. Serverul de mesaje, care poate fi de asemenea denumit și *router* de mesaje, sau *broker* de mesaje este responsabil cu livrarea mesajelor între clienții sistemului de mesagerie. Serverul de mesaje realizează, în acest fel, decuplarea clientului care produce mesajul de cel care consumă respectivul mesaj. Clienții intră în contact numai cu serverul de mesaje și nu cu ceilalți clienți ai sistemului, permițând eliminarea sau adăugarea de noi clienți, fără afectarea sistemului ca un întreg. În mod normal, o arhitectură centralizată utilizează o topologie de tip stea. În configurația cea mai simplă, avem de aface cu un broker de mesaje plasat central, la care se conectează toți clienții sistemului. Arhitectura de tip stea oferă un număr minim de conexiuni de rețea, interconectând totodată fiecare componentă 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 observație, o arhitectură MOM centralizată poate furniza un model generic de mesagerie pentru sisteme extreme de complexe, oferind un grad înalt de flexibilitate și posibilități facile de redimensionare.

Arhitecturile descentralizate folosesc la nivel de rețea protocolul *IP multicast*. Un sistem de mesagerie bazat pe multicasting nu are un server central pentru livrarea mesajelor. Unele dintre funcționalitățile care ar fi oferite de server sunt încorporate în partea de client (persistare, facilități de lucru cu tranzacții, aspecte legate de siguranța comunicării), în timp ce rutarea mesajelor este delegată către nivelul de rețea, prin folosirea protocolului *IP multicast*. Acest protocol oferă posibilitatea aplicațiilor de subscrie la unul sau la mai multe grupuri de *multicast*. Fiecare grup de *multicast* folosește o adresă IP de rețea prin care fiecare mesaj ce este primit va fi redistribuit către toți membrii grupului. În acest fel, aplicațiile pot trimite mesaje către o adresă de IP de tip *multicast* și se așteaptă ca nivelul de rețea 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 rețea, fără a mai fi necesar un server dedicat. Cu toate acestea, așa cum am menționat inițial, anumite funcționalități oferite de server vor fi necesar a fi incluse în fiecare client, aspect ce poate conduce la încărcarea clienților și la distribuirea de funcționalități care ar fi mai simplu de gestionat într-un singur loc, cel puțin la nivel logic.

9.2.2. Modele de mesagerie furnizate de Java Message Service (JMS)

Cu toate că interfața pusă la dispoziție de *Java Message Service (JMS)* nu s-a modificat semnificativ de la introducerea ei în 1999, maniera în care soluțiile *middleware* orientat pe mesaje sunt utilizate în sisteme și aplicații diverse s-a schimbat semnificativ. Sistemele de mesagerie sunt astăzi utilizate pentru a rezolva probleme legate de siguranța livrării datelor între aplicații, respectiv probleme legate de capacitatea de adaptare dinamică în funcție 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 și subscriere (*publish-and-subscribe/pub-sub*).

Principial, modelul *pub/sub* presupune publicarea mesajelor de la unul către mai mulți (*broadcast*), în timp ce modelul *p2p* este dedicat traserii de mesaje unu-la-unu (Figura 9.1).

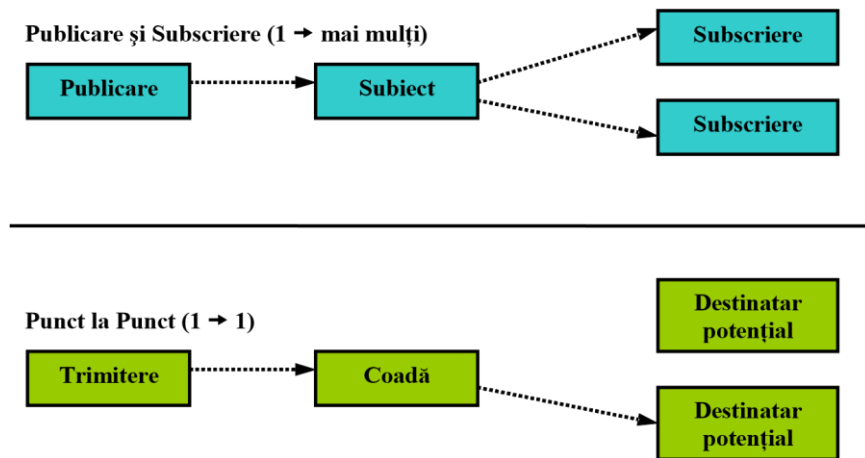


Figura 9.1. Modelele de mesagerie furnizate de JMS API

Din perspectiva JMS, clienții conectați la platforma de mesagerie se numesc *clienți JMS*, iar sistemul de mesagerie, cel care furnizează serviciul de distribuire a mesajelor între clienți, se numește *JMS provider*. În general, o aplicație JMS este un sistem informatic care este compus din mai mulți clienți JMS și un server JMS. În acest context, un client JMS care trimite un mesaj în sistem se numește producător al acelui mesaj, în timp ce un client JMS care primește un mesaj din sistem se numește consumator al respectivului mesaj. Trebuie remarcat faptul că un client JMS poate fi în același timp și producător și consumator de mesaje.

a) Modelul de comunicare punct-la-punct (*p2p*)

Acest model permite clienților JMS să trimită și să primească mesaje, atât în manieră sincronă cât și în manieră asincronă, prin intermediul unui canal de comunicație 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 tradițional, modelul *p2p* este un model în care mesajele sunt în mod activ cerute din coadă (*pull-based* sau *poll-based*) de către aplicația căreia îi sunt destinate, în loc să fie făcute disponibile către aplicație imediat ce sunt recepționate (*pushed*). Depinzând de implementare, chiar și în modelul *p2p*, serverul de mesaje poate însă să facă disponibile mesajele din coadă (*push*) aplicației 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 aplicație client și numai una, chiar dacă pot asculta mai multe aplicații client la o aceeași coadă de mesaje.

Acest model oferă două mecanisme de transmitere a mesajelor la nivel de aplicație client:

- trimitere de mesaj fără a aștepta 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 comunicație numit *topic*. Aplicațiile care produc mesajele trimise către acest canal se numesc aplicații care publică mesaje, în timp ce aplicațiile care consumă mesaje de la un anumit *topic* se numesc aplicații care subscriu la acel *topic*. Spre deosebire de modelul *p2p*, mesajele publicate la un *topic* pot fi primite de mai multe aplicații care subscriu la acel *topic*. Această tehnică este uneori numită *broadcast* și reprezintă modalitatea prin

care un mesaj este distribuit către toate aplicațiile interesate în recepționarea 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 dispoziția aplicației 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 aplicațiile implicate. Clientul serverului de mesagerie care publică mesaje nu are, în general, cunoștință de câți clienți care consumă acele mesaje au subscris la *topic* și ce anume intenționează aceștia să facă cu aceste mesaje. De exemplu, o aplicație care publică date legate de tranzacțiile realizate de către o instituție bursieră, nu trebuie în mod necesar să știe câți clienți au subscris la aceste date și felul în care acești clienți prelucrează datele avute în vedere.

Din perspectiva clienților 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 aplicația client primește mesajele care sunt publicate la subiectul la care a subscris numai pe durata existenței 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 aplicațiile implicate, decât modelul *pub/sub*. De exemplu, un trading GUI poate trimite un ordin către o coadă pentru cumpărarea sau vânzarea unui instrument financiar și apoi să aștepte pentru un răspuns de confirmare, conținând identificatorul unic în sistem al ordinului, generat de 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 aplicațiilor care consumă și prelucrează mesajele trimise către o coadă. În acest sens se pot lansa în mod dinamic mulți consumatori ai mesajelor unei aceleiași cozi, multiple instanțe ale aceleiași aplicații, putându-se construi o logică de distribuire optimă a mesajelor cozii între aceste instanțe. Trebuie menționat faptul că specificațiile oferite de JMS nu includ reguli privind distribuirea mesajelor între mai multe aplicații destinatar, însă anumite produse comerciale au ales să implementeze abilități privind echilibrarea încărcării consumatorilor. Față de modelul *pub/sub*, modelul *p2p* oferă posibilitatea aplicațiilor client de a consulta coada de mesaje (*queue browser*) și vizualiza conținutul mesajelor înainte de a le consuma (extrage din coadă).

9.2.3. Interfețe de comunicare JMS

JMS este o interfață de comunicare orientată pe mesaje care a fost creată de către Sun Microsystems pe baza specificațiilor JSR-914 (*Java Specification Request*) în 2002. JMS nu este un sistem de mesagerie în sine, ci o interfață abstractă împreună cu clasele necesare clienților unui sistem de mesagerie pentru ca aceștia să poată comunica cu sistemul. Utilizând interfața JMS, o aplicație client a unui sistem (server) de mesagerie devine portabilă, din perspectiva serverului concret de mesagerie la care se conectează.

Crearea interfeței JMS a fost un efort al întregii industrie software. Obiectivul inițial a fost acela de a furniza un API Java pentru conexiunea la sisteme de mesagerie deja existente la acel moment. Ulterior, scopul inițial 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 existența sistemelor bazate pe RPC (*Remote Procedure Call*), precum CORBA și Enterprise JavaBeans.

Interfața JMS este formată din trei componente principale:

- partea generală a interfeței;
- interfața *p2p*;
- interfața *pub/sub*.

Interfața *p2p* este utilizată exclusiv pentru modelul de mesagerie bazat pe cozi, în timp ce interfața *pub/sub* este destinată utilizării exclusive a modelului de mesagerie bazat pe subiecte (*topics*). API-ul general este compus din șapte interfețe dedicate trimiterii și recepționării de mesaje:

ConnectionFactory, Destination, Connection, Session, Message, MessageProducer, MessageConsumer

Din aceste șapte interfețe generice (Figura 9.2), conform specificațiilor JMS, *ConnectionFactory* și *Destination* trebuie să fie obținute de la serverul de mesagerie, utilizând JNDI (*Java Naming and Directory Interface*).

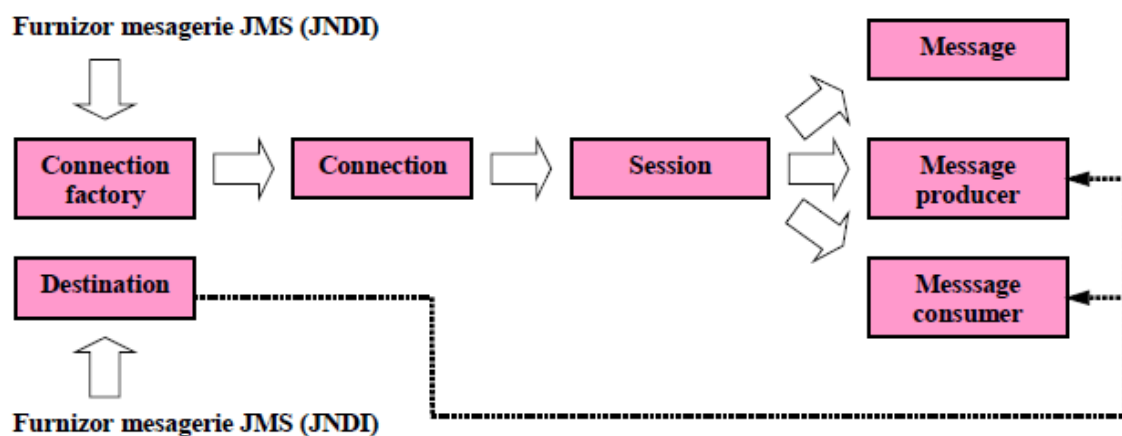


Figura 9.2. Interfețe generice JMS API

Celelalte interfețe sunt create prin diversele metode puse la dispoziție de JMS API. De exemplu, odată ce avem creat *ConnectionFactory*, putem instanția un obiect de tip *Connection*. Odată ce avem un obiect de tip *Connection*, putem crea unul de tip *Session*, iar mai departe, în cadrul instanței *Session* putem crea obiecte de tip *Message*, *MessageProducer* și *MessageReceiver*.

În JMS, obiectul *Session* deține partea tranzacțională a lucrului cu mesaje și nu obiectul *Connection*, așa cum este cazul interfeței JDBC. Aceasta înseamnă că, atunci când utilizează JMS, o aplicație va avea în mod normal numai o singură instanță de tip *Connection*, însă poate deține o colecție de obiecte de tip *Session*.

Există, de asemenea, în cadrul JMS API, o multitudine de interfețe care sunt destinate gestionării excepțiilor, priorității mesajelor și manierei în care acestea sunt persistate. Interfața JMS pentru modelul de comunicare *p2p* se referă în mod particular la mesageria bazată pe cozi de mesaje. Interfețele utilizate pentru trimiterea și primirea de mesaje în cadrul acestui model sunt următoarele: *QueueConnectionFactory, Queue, QueueConnection, QueueSession, Message, QueueSender, QueueReceiver*.

Întreaga ierarhie de interfețe și maniera prin care acestea interacționează unele cu altele sunt similare modelului general. La nivel sintactic, reflectat până la urmă și în semantica fiecărei interfețe, conceptul

generic de destinație este înlocuit cu cel de coadă (Figura 9.3.). Aplicațiile care utilizează modelul de mesagerie *p2p* vor folosi în mod normal interfețele bazate pe cozi în locul interfețelor generice

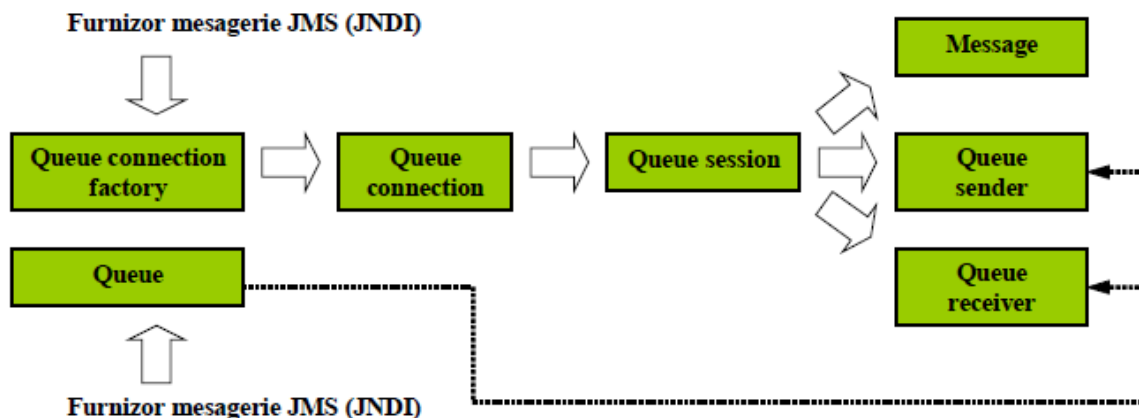


Figura 9.3. Interfețe JMS pentru modelul de comunicare bazat pe cozi de mesaje

Modelul de comunicare pub/sub este oferit de JMS în aceeași manieră standardizată de prezentare a interfețelor. La nivel sintactic cuvântul *Queue* este înlocuit de cuvântul *Topic*. Interfețele pentru modelul *pub/sub* sunt următoarele:

TopicConnectionFactory, Topic, TopicConnection
TopicSession, Message, TopicPublisher, TopicSubscriber .

Singurele excepții, din perspectiva numelui, sunt reprezentate de interfețele *TopicPublisher* și *TopicSubscriber*. Însă, așa cum subliniam mai devreme, JMS API este foarte intuitiv în ceea ce privește

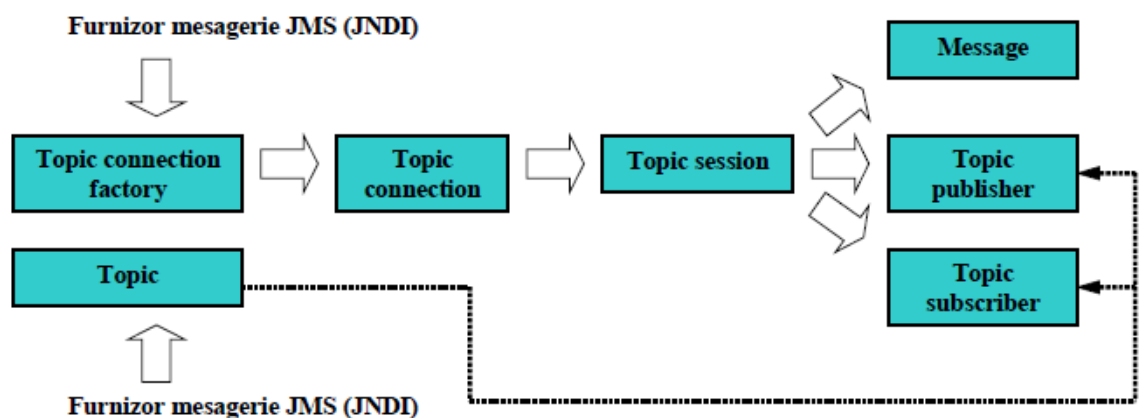


Figura 9.4. Interfețe JMS pentru comunicarea în modelul pub/sub

denumirea și utilizarea interfețelor pe care le pune la dispoziție. Din acest punct de vedere, modelul *pub/sub* utilizează *topics* împreună cu *publishers* și *subscribers*, acolo unde modelul *p2p* folosește *queues* împreună cu *senders* și *receivers*. Așa cum este reiese și din Figura 9.4 terminologia și relațiile între interfețele oferite pentru canalizarea mesajelor în cadrul modelului *pub/sub* se mulează fidel pe cele ale API-ului generic oferit de JMS.

9.2.4. Structura unui mesaj JMS

Vom prezentării structura internă a unui mesaj JMS, respectiv a părților care compun mesajul:

- componentele de antet (*headers*) ;

- atributele mesajului (*properties*) ;
- diverse tipuri de date utile manipulate de aplicație (*message payloads*).

| |
|--|
| Headers (antet mesaj) JMSTDestination JMSDeliveryMode JMSMessageID JMSTimestamp JMSExpiration JMSRedelivered JMSPriority JMSReplyTo JMSCorrelationID JMSCorrelationIDAsBytes JMSType |
| Properties (atribute mesaj) |
| Payload (date de business specifice aplicației) |

Clasa *Message* reprezintă cea mai importantă parte din specificațiile care stă la baza JMS. Toate datele și evenimentele utilizate de o aplicație fundamentată pe JMS sunt comunicate prin mesaje, celelalte componente ale interfeței JMS au menirea de a facilita transferul acestor mesaje. Un mesaj JMS joacă un rol unic în domeniul sistemelor distribuite. El poate transporta atât date specifice de aplicație, cât și notificări pentru evenimente legate de activitatea aplicațiilor sistemului. Un mesaj JMS nu este o comandă, prin el se transferă date și se poate informa aplicația care l-a recepționat că a avut loc un anumit eveniment. Un astfel de mesaj nu îi impune aplicației care îl recepționează ce anume să facă cu el, iar aplicația care l-a produs nu așteaptă după un răspuns. Se obține în acest fel o decuplare a producătorului mesajului de consumatorul acestuia, făcând sistemele orientate pe mesaje mult mai dinamice și mai flexibile decât cele bazate pe paradigma cerere/răspuns realizate în manieră sincronă. Mesajele JMS pot fi de mai multe tipuri, în funcție de maniera în care sunt organizate datele utile de transferat (*payload*), respectiv pot fi foarte structurate, cum sunt *StreamMessage* și *BytesMessage*, sau aproape deloc structurate, cum sunt tipurile *TextMessage*, *ObjectMessage* și *MapMessage*. Așa cum am menționat mai sus, în toate cazurile, mesajele pot transporta atât notificări cât și date de business.

Antetul unui mesaj JMS conține date globale ce descriu cine sau ce aplicație a creat mesajul, când a fost acesta creat, cât timp urmează să fie valide datele de business pe care le conține etc. Antetul conține, de asemenea, informații de rutare a mesajului, care descriu destinația acestuia (*topic* sau *queue*), cum trebuie să se realizeze confirmarea de primire a acestuia ș.a. Fiecare mesaj JMS conține 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 componentele antetului unui mesaj JMS, ele făcând parte din descrierea interfeței *Message*:

Elementele de antet JMS sunt grupate în două categorii: elemente asigurate automat și elemente asigurate de programator.

Elementele de antet asigurate automat

Majoritatea antetelor mesajelor JMS sunt asigurate automat. Valorile elementelor acestor antete sunt setate de serverul de mesagerie atunci când mesajul este livrat, astfel încât valorile asigurate de

programator prin utilizarea metodelor *setJMS<HEADER>()* sunt ignorate. Cu toate acestea, unele elemente de antet asigurate automat pot fi setate în mod programatic atunci când se instanțiază clasele *Session* și *MessageProducer* (respectiv *TopicPublisher*). În această situație se găsesc elementele de antet *JMSDeliveryMode* și *JMSPriority*.

JMSDestination - Prin acest element de antet se identifică destinația mesajului, care poate fi fie un obiect de tip *Topic* fie unul de tip *Queue*, ambele derivate din tipul *Destination*. Identificarea destinației mesajului este importantă, mai ales pentru clienții 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 și nepersistent. Un mesaj persistent este livrat o dată și numai o singură dată, ceea ce înseamnă că dacă serverul de mesagerie înregistrează o disfuncționalitate mesajul nu este pierdut; el va fi livrat după ce serverul își 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 eșuează în funcționare. În ambele moduri de livrare, persistent și nepersistent, serverul de mesagerie nu trebuie să trimită un mesaj către un același consumator mai mult de o dată (lucru posibil totuși 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 dorește să arhiveze toate mesajele pe care le-a primit și 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 corespondență 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* conține timpul la care mesajul a fost recepționat de serverul de mesagerie și 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 și 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 preveni livrarea către consumatori a unui mesaj care nu mai este de actualitate, din perspectiva informației 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 viață la 1 oră (1000 milisecunde x 60 secunde x 60 minute)
topicPublisher.setTimeToLive(3600000);
```

Serverul de mesagerie va adăuga valoarea *timeToLive* la timpul sistemului și 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 dorește a fi retrimis către consumator. Este modelat ca o valoare booleană, când *JMSRedelivered* este *true* atunci mesajul este retransmis, când este *false* nu se realizează acest lucru. Un mesaj poate fi marcat pentru retransmitere atunci când consumatorul a eșuat să confirme primirea acestuia la livrarea anterioară, 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 priorități 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 urgență/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 urgență/alertă au întâietate față 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 totuși 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 aplicația care va consuma mesajul să trimită înapoi un mesaj de răspuns (de exemplu atunci când se folosește un mecanism de cerere/răspuns prin intermediul platformei de mesagerie). Elementul de antet *JMSReplyTo*, care conține un obiect de tip *javax.jms.Destination*, indică care este adresa la care consumatorul JMS va trebui să trimită răspunsul. Utilizarea acestei priorități 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 opțional al antetului, care este setat de clientul JMS. Menirea lui este aceea de a identifica structura mesajului și 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 acționează ca elemente de caracterizare suplimentară a mesajului, pe lângă cele conținute de antet. Ele permit programatorului să adauge mai multe informații structurate cu privire la conținutul 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 dorește filtrarea acestora. Interfața *Message* oferă mai multe metode de acces și 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 aplicației; atribute definite de către interfața JMS; atribute specifice serverului de mesagerie.

Java Message Service definește șase tipuri de interfețe *Message* care trebuie să fie oferite de furnizorii de mesagerie compatibilă JMS. Implementare acestor interfețe este însă lăsată la latitudinea producătorilor. Cele șase interfețe sunt formate din interfața *Message* și cinci subinterfețe ale sale:

TextMessage, *StreamMessage*, *MapMessage*, *ObjectMessage*, *BytesMessage*

Message - Tipul cel mai simplu de mesaj JMS este modelat de *javax.jms.Message* și servește și ca interfață de bază pentru celelalte interfețe. Tipul *Message* poate fi creat și folosit ca mesaj JMS și fără a purta conținut de business (*payload*). Acest tip de mesaj conține numai antetul și atributele (proprietățile) 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 și livrare ulterioară garantată, furnizează funcționalități care oferă aplicațiilor pe care le deservește posibilitatea de a rula într-o manieră continuă, fără întreruperi pentru mentenanță. Oferă flexibilitate integrării de noi componente prin furnizarea unor modele de comunicare de tip *pub/sub* și *p2p* într-o manieră asincronă. Datorită transparenței oferite în ceea ce privește localizarea componentelor și a mecanismelor de administrare și control, furnizează premisele unei arhitecturi 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 că 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 rețea LAN sau WAN
- Intrarea în și 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 înțelege 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 trimite/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 suntificați 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. În momentul în care o aplicație client dorește 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 îl transmite spre stiva de protocoale, care îl transmite către protocolul din vârful stivei. Protocolul procesează mesajul și îl transmite corespondent în stivă către protocolul de transport și acesta îl 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 dorește 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. Fiecare canal menține o listă cu procesele care sunt conectate la același 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 și același nume de cluster vor face parte din același cluster. În 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()` – invocată de un nod pentru a părăsi cluster-ul
- `close()` – distruge canalul. Un canal închis nu mai poate fi redeschis. În cazul în care canalul nu este deconectat de la cluster, realizează și deconectarea acestuia
- `getLocalAddress()` – returnează adresa locală a nodului

Un view este o reprezentare a nodurilor prezente la un moment dat în cluster. În 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 apelată în momentul recepționării unui mesaj
- `viewAccepted()` – este apelată când un nod se alătură sau părăsește cluster-ul

Structura unui mesaj este prezentată în continuare. Acesta conține adresa sursă, 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. Aplicație de chat bazat pe modelul pub-sub din JMS

Aplicația este destinată comunicării de mesaje între mai mulți clienți în sistem tip chat, folosind modelul *publish/subscribe*. Aceasta folosește pachetul Java JMS pentru transmisia și recepția mesajelor, pasarea acestora fiind intermediată de broker-ul JMS care trebuie să ruleze ca serviciu.

Aplicația este compusă din următoarele module:

AplicatieChat – conține funcția *main()* și codul de inițializare al ferestrei,
PanouChat – panou care conține controalele folosite la afișarea mesajelor, *dialogInfoConexiune* – căsuța de dialog care permite selectarea parametrilor de conectare, interfața *CreatorMesaje*, și clasele ce implementează această interfață – *CreatorMesajText* și *CreatorMesajStream*, folosite la crearea mesajelor de tip text și stream, încapsulând informații care identifică unic mesajul cum ar fi expeditorul și tipul acestuia.

Pentru trimiterea mesajelor, aplicația folosește clase derivate din clasa *Message* a pachetului JMS – *TextMessage* și *StreamMessage*.

Tipurile de mesaje suportate sunt:

- INTRARE – intrare în sesiunea de chat,
- NORMAL – mesaj propriu-zis,
- IESIRE – ieșire din sesiunea de chat.

Fiecare mesaj conține, pe lângă textul propriu-zis și câmpuri care identifică expeditorul și tipul mesajului. Codificarea acestor câmpuri se face fie folosind proprietăți ale clasei *Message* (*CreatorMesajText*), fie codificând aceste câmpuri în cadrul unui stream (*CreatorMesajStream*).

Când aplicația intră într-o sesiune, transmite un mesaj broadcast INTRARE. Toți clienții aflați în sesiune vor primi acest mesaj și fereastra va afișa intrarea noului client. Odată ce aplicația a intrat în sesiune, mesajele trimise sunt codificate ca obiecte *Message*. La transmiterea acestor mesaje (publicare), toți clienții vor afișa în fereastra de mesaje numele expeditorului și mesajul. Când un client se deconectează, se transmite un mesaj IESIRE.. Pentru încapsularea numelui expeditorului și a tipului mesajului, s-au folosit două metode de codificare: clasa *CreatorMesajText* folosește proprietățile puse la dispoziție de clasa JMS *TextMessage*, asociind valoarea corespunzătoare unui nume de proprietate, clasa *CreatorMesajStream* scrie cele două informații direct în streamul de ieșire, după acestea fiind scris mesajul efectiv.

La evenimentul de primire a unui mesaj, citirea trebuie făcută în aceeași 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 menționate, mesajul rezultat fiind trimis efectiv prin publicarea acestuia folosind un obiect JMS de tip *TopicPublisher*, prin metoda *publish()*.

Pentru ca și clienții să primească mesajele trimise în sesiunea de chat, aceștia trebuie să-și creeze un obiect de tip *TopicSubscriber*, pentru care trebuie să stabilească o clasă ce implementează interfața JMS *MessageListener*. Aceasta clasă ascultă mesajele primite din sesiune definind metoda *onMessage()* în care aplicația preia mesajul și îl afișează în interfață.

Succesiunea de operații 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 pași, clientul își anunță prezența în sesiune trimițând un mesaj broadcast INTRARE.

Folosirea tehnologiei JMS a permis crearea unei aplicații în care clienții comunică prin mesaje, publicându-le într-un sistem unificat. Între clienți cuplarea este slabă, aceștia operând doar cu un obiect de tip sesiune care se ocupă de gestionarea trimiterii mesajelor în mod asincron, nefiind necesare mecanisme de confirmare a recepției. Broker-ul JMS se ocupă de funcționarea sesiunilor existente în aplicațiile care utilizează această tehnologie, existând o conexiune permanentă cu acesta.

Mesajele trimise se pot codifica în diverse formate, cu ajutorul claselor puse la dispoziție de pachetul JMS în acest scop. Dintre acestea amintim: text simplu, stream, șir de octeți, mesaje care încapsulează obiecte etc. Aplicația funcționează în sistem *publish/subscribe*, în care fiecare client publică în sesiune mesaje de diferite tipuri, acestea fiind recepționate de ceilalți clienți conectați și interpretate în funcție de tip (mesaje de intrare în sesiune, de ieșire din sesiune și mesaje propriu-zise). Obiectul JMS care identifică un grup de discuții este definit de clasa *Topic*, cu ajutorul căruia mesajele trimise sunt direcționate doar către clienții care sunt conectați la același grup, făcându-se efectiv o distribuire a acestora în funcție de topic.

9.3.2. Aplicație SimpleChat implementată cu tehnologia JGroups

Simple Chat este o aplicație 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 alătură grupului. La execuție se creează 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. În 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. În 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 creează 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 efectuată 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();
            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) {} } }

```

În 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 destinție nulă înseamnă că mesajul va fi trimis spre toate nodurile din cluster. Dacă adresa destnație nu e nulă, mesajul este trimis doar spre acea adresă. Al doilea argument este adresa la care se trimite mesajul. Stiva de protocol va insera în 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 prindează 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 într-un cluster. De exemplu 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 și 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 destinatarul (fără a avea cunoștințe referitor la aceștia), respectiv existența unor destinatari sau abonați (*subscribers*) care să primească acele mesaje trimise de expeditori (*publish-eri*), pentru care au manifestat interes înregistrându-se/abonându-se la aceștia. 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 (*subscribers*), 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 entități 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 fiabilitatea. Astfel, în contextul MOM, aplicațiile ar putea să nu reușească să proceseze mesaje 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 mesaje, iau un mesaj și îl direcționează către zero sau mai multe cozi. Algoritmii 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 <code>amq.direct</code> |
| Fanout exchanger | <code>amq.fanout</code> |
| Topic exchanger | <code>amq.topic</code> |
| Headers exchanger | <code>amq.match</code> (și <code>amq.headers</code> 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 foarte 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 alte 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-ului 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 existe 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 încheie conexiunea AMQP 0-9-1 în locul închiderii bruște a conexiunii 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 și clienții îl folosesc pentru a determina pentru care canal va fi procesată operația/metoda.

Un canal există doar în contextul unei conexiuni și niciodată pe cont propriu. Când o conexiune este închisă, toate canalele de pe ea sunt închise. Pentru aplicațiile care utilizează mai multe thread-uri/procese, se obișnuiește 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 și oferă medii complet izolate în care trăiesc entitățile AMQP. Clienții AMQP specifică ce vhosts doresc să utilizeze în timpul „negocierii” conexiunii.

10.2.2. Frameworkul RabbitMQ

RabbitMQ (www.rabbitmq.com) este o implementară cu largă utilizare și performantă a standardului AMQP, ce conține complementar o serie de extensii, astfel:

- **Stocare în memorie și durabilitate opțională (limitată)**

RabbitMQ stochează toate mesajele pe care le manipulează în memorie, oferind în același 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 și, ca atare, nu permite consumatorilor să refolesească 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 înregistrați este suficientă pentru ca broker-ul să înlătoreze mesajul. Totuși, mesajele de confirmare ajută la reducerea numărului de mesaje neprocesate, oferind în același 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 și, prin urmare, împiedică modificarea acesteia, în timp ce conexiunea cu RabbitMQ nu întâmpină niciun eșec, aceasta caracteristică se poate dovedi a fi folositoare.

- **Cozi și exchangeri declarați dinamic**

Chiar dacă administratorul broker-ului le poate crea în avans, noi cozi și exchangeri pot fi creați direct în codul producătorilor și consumatorilor. Aceasta este o practică foarte bună pentru a se asigura că mesajele nu sunt trimise către cozi și exchangeri inexistenți.

- **Gestionarea permisiunilor și suport SSL**

RabbitMQ le cere producătorilor, consumatorilor și administratorilor să se autentifice înainte de a putea efectua orice operațiune pe broker. Pot fi acordate diferite permisiuni diferiților utilizatori, permițând astfel aplicarea unui control eficient al accesului. SSL este, de asemenea, acceptat și poate fi utilizat pentru autentificare și pentru criptarea datelor transmise prin canale între broker și producători și 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 cei care vor vizita platforma – clienții. Aplicația permite oricărui client să devină abonat al platformei atunci când acesta, prin intermediul interfeței 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 într-un 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 urmărirea pașilor 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 execuția modului 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. De asemenea, 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:

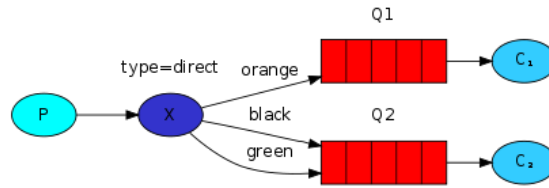


Figura 10.1. Configurație Uni-binding

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 direcționate 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 configurația expusă mai jos:

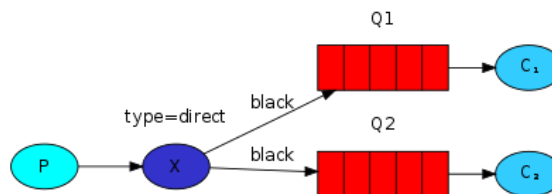


Figura 10.2. Configurație Multi-binding

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 emițăorul 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 severitatea logging-ului: info, warning, error. Astfel, configurația finală a aplicației ar trebuie să fie următoarea:

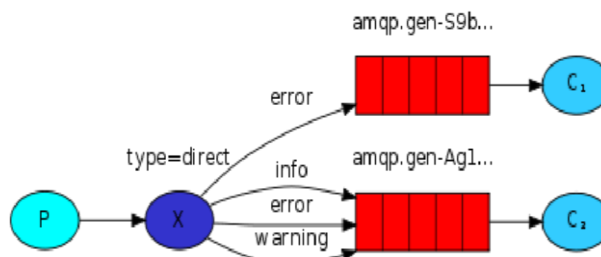


Figura 10.3. Configurația aplicației de logging

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.

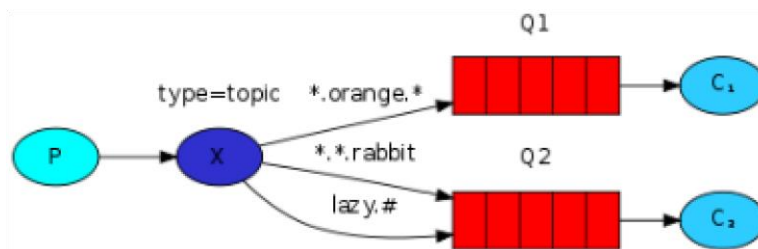


Figura 10.4. Configurație aplicație ce consumă selectiv mesaje (topicuri)

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 summarize 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 **EmitLogTopic.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 critical: 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 sunt acestea realizate ?

10.4.3. Explicați și argumentați diferențele conceptuale între coadă și canal. Explicați modelul de mesageriepub-sub și particularitățile sale în tehnologia RabbitMQ

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:

- Trei consumatori și o singură coadă;
- Patru consumatori, trei cozi și doi exchanger-i;
- 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;
- 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 11.2 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.

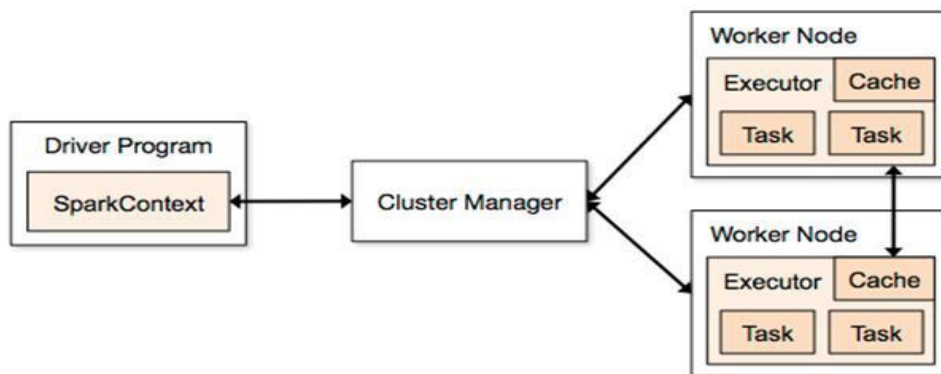


Figura 11.2. Cluster Spark [1]

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 recuperare 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 rețin 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 utilizatorul 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 dependențe cu RDD-ul părinte, o funcție de calcul și metadata 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 e fișiere

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 dependențe cu RDD-urile părinte. O transformare poate fi efectuată pe unele RDD-uri, care reprezintă diferite fișiere HDFS, iar output-ul transformării 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 dependențele cu RDD-ul părinte în două tipuri: înguste (narrow) și largi (wide). Dependențele înguste indică faptul că fiecare partiție fiu RDD depinde de un număr fix de partiții din părintele RDD, cum ar fi transformările map. Dependențele 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 dependențe înguste pot fi compuse într-un singur cluster, cum ar fi operația map urmată de operația filter. În mod contrariu, dependențele 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 dependență 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 dependențelor largi. Această clasificare asigură 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 dependențe înguste. O nouă etapă începe atunci când există o transformare cu dependențe 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: filter, count, distinct, sample) sau mai mare (transformări precum: flatMap(), union(), cartesian()) sau de aceeași 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), intersection(other-dataset), distinct(), groupByKey(), reduceByKey(func, [numTasks]), sortByKey(), join(), coalesce().

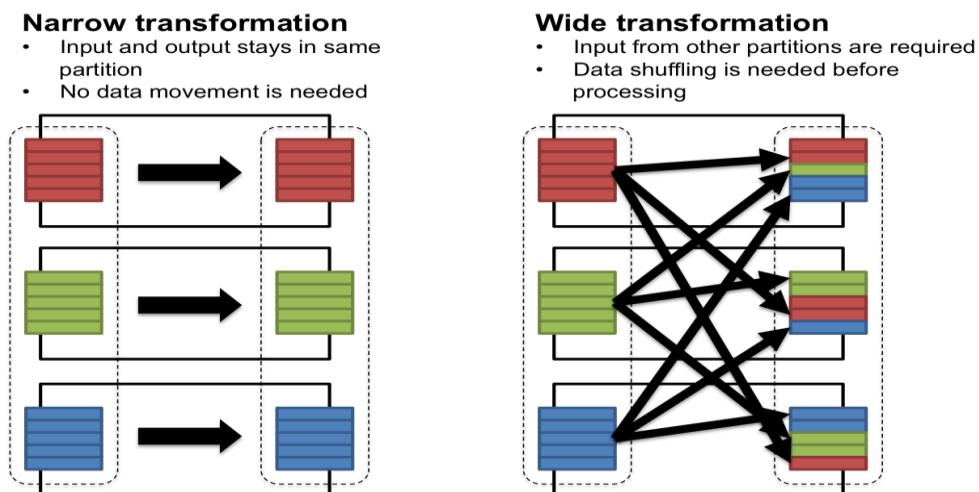


Figura 11.3. Transformări Narrow vs. Wide [2]

Acțiuni

Transformările permit construirea unui plan specific de transformare logică. Pentru a declanșa calculul, se execută o acțiune. Acțiunea instruește 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 introducerea 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.

Stream-urile de date continue de intrare sunt reprezentate ca fluxuri discretizate (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:

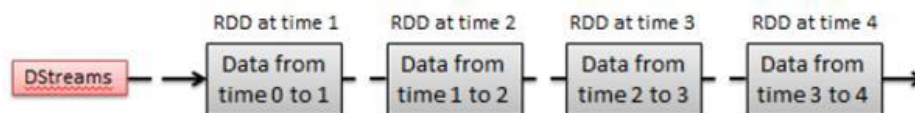


Figura 11.5.Reprezentarea modului în care DStream-urile sunt partiționate în Resilient Distributed Datasets

Caracteristici Apache Spark Streaming:

- *Fault tolerance*- datorită RDD ce reprezintă sunt seturi de elemente de date imutabile care pot fi re-calculat 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 acțiuni 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) |

| | |
|---|---|
| <p>reduceByKey(func,[numTasks])</p> <p>Purpose: : To aggregate values of a key using a function. “numTasks” is an optional parameter to specify number of reduce tasks</p> | <pre>scala> val word1=fm.map(word=>(word,1)) scala> val wrdCnt=word1.reduceByKey(_+_)</pre> <p>scala> wrdCnt.collect()</p> <p>Result: Array[(String, Int)] = Array((is,2), (It,1), (awesome,1), (Spark,1), (fun,1))</p> |
| <p>groupByKey([numTasks])</p> <p>Purpose: To convert (K,V) to (K,Iterable<V>)</p> | <pre>scala> val cntWrd = wrdCnt.map{ case (word, count) => (count, word) } scala> cntWrd.groupByKey().collect()</pre> <p>Result: Array[(Int, Iterable[String])] = Array((1,ArrayBuffer(It, awesome, Spark, fun)), (2,ArrayBuffer(is)))</p> |
| <p>distinct([numTasks])</p> <p>Purpose: Eliminate duplicates from RDD</p> | <pre>scala> fm.distinct().collect()</pre> <p>Result: Array[String] = Array(is, It, awesome, Spark, fun)</p> |

Tabel 11.2.Operatori uzuali pe seturi de date (din [6])

| Transformarea și scopul | Exemplu și Rezultat |
|--|--|
| <p>union()</p> <p>Purpose: new RDD containing all elements from source RDD and argument.</p> | <pre>Scala> val rdd1=sc.parallelize(List('A','B')) scala> val rdd2=sc.parallelize(List('B','C')) scala> rdd1.union(rdd2).collect()</pre> <p>Result: Array[Char] = Array(A, B, B, C)</p> |
| <p>intersection()</p> <p>Purpose: new RDD containing all elements from source RDD and argument.</p> | <pre>Scala> rdd1.intersection(rdd2).collect()</pre> <p>Result: Array[Char] = Array(B)</p> |
| <p>cartesian()</p> <p>Purpose: new RDD cross product of all elements from source RDD and argument.</p> | <pre>Scala> rdd1.cartesian(rdd2).collect()</pre> <p>Result: Array[(Char, Char)] = Array((A,B), (A,C), (B,B), (B,C))</p> |
| <p>subtract()</p> <p>Purpose: new RDD created by removing data elements in source RDD in common with argument</p> | <pre>scala> rdd1.subtract(rdd2).collect()</pre> <p>Result: Array[Char] = Array(A)</p> |
| <p>join(RDD,[numTasks])</p> <p>Purpose: When invoked on (K,V) and (K,W), this operation</p> | <pre>scala> val personFruit = sc.parallelize(Seq(("Andy", "Apple"), ("Bob", "Banana"), ("Charlie", "Cherry"), ("Andy", "Apricot"))) scala> val personSE = sc.parallelize(Seq(("Andy",</pre> |

| | |
|--|--|
| creates a new RDD of (K, (V,W)) | <pre> “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))) </pre> |
| cogroup(RDD,[numTasks]) Purpose: To convert (K,V) to (K,Iterable<V>) | <pre> 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)))) </pre> |

Tabel 11.3. Acțiuni uzuale (din [6])

| Acțiunea și scopul | Exemplu și Rezultat |
|--|--|
| count() Purpose: Get the number of data elements in the RDD | <pre> scala> val rdd = sc.parallelize(List('A','B','C')) scala> rdd.count() Result: Long = 3 </pre> |
| collect() Purpose: get all the data elements in an RDD as an array | <pre> scala> val rdd = sc.parallelize(List('A','B','C')) scala> rdd.collect() Result: Array[Char] = Array(A, B, C) </pre> |
| reduce(func) Purpose: Aggregate the data elements in an RDD using this function which takes two arguments and returns one | <pre> scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.reduce(_+_) Result: Int = 10 </pre> |
| take (n) Purpose: : fetch first n data elements in an RDD. Computed by driver program. | <pre> Scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.take(2) Result: Array[Int] = Array(1, 2) </pre> |
| foreach(func) Purpose: execute function for each data element in RDD. Usually used to update an accumulator(discussed later) or interacting with external systems. | <pre> 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 </pre> |
| first() | <pre> scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.first() </pre> |

| | |
|---|---|
| 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/gutenberg.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/stevelloughran/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:

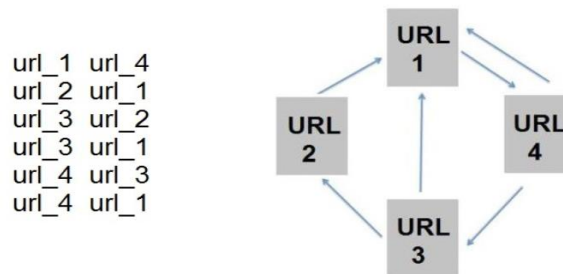


Figura 11.8. Exemplu page rank

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.

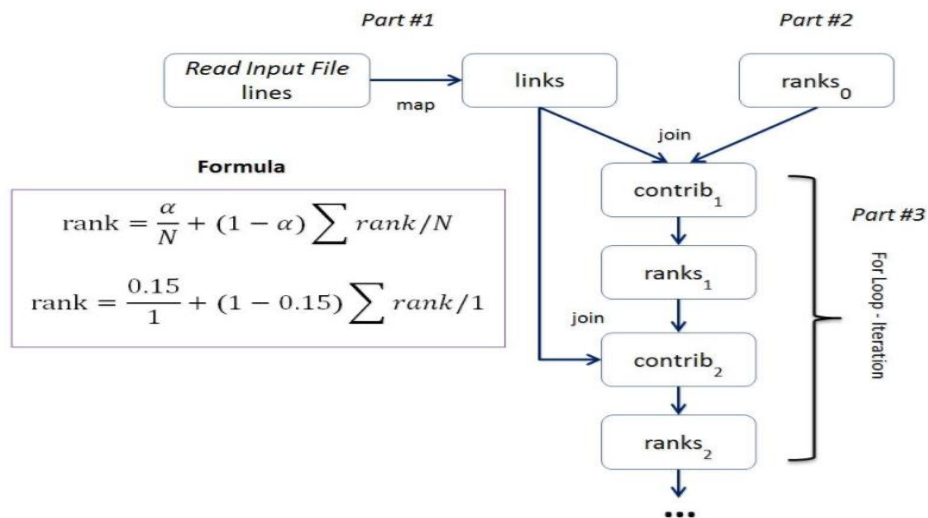


Figura 11.9. Modelul de execuție PageRank

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 să 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.

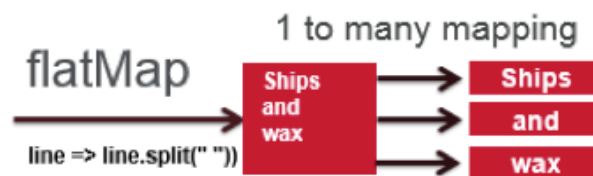


Figura 11.10. Transformarea flatMap

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 reduceByKey(func). Se va returna un RDD care conține pereche (cuvânt, număr de apariții).

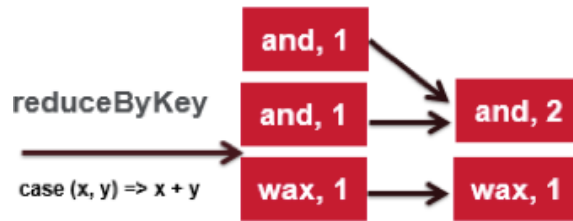


Figura 11.11.Transformarea reduceByKey

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.

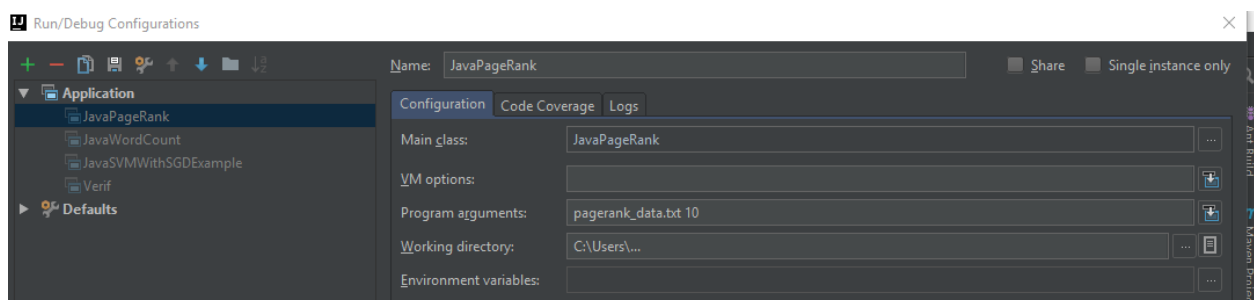
Stochastic gradient descent (SGD) este o aproximare randomizată a algoritmului de gradient descent (batch) pentru a minimiza o funcție obiectiv diferențiată continuă. În învățarea automată supravegheată, funcția obiectiv este o funcție de pierdere - loss function(logistică, suma pătratelor, etc.). În 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 parametri pentru rulare. Pentru a seta acești parametri 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 asigură 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 SparkContext 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 parametri definesc proprietăți pentru aplicația Spark, iar alții sunt folosind pentru alocarea de resurse în 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 clusterul, 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 aceiași 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 și 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 constate 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 numărului 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 într-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 structurează conținutul de date 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 algoritmi 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 algoritmi “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

| | |
|---|-----|
| Tabel 4.1.Moduri de comunicare la baza primitivelor MPI | 41 |
| Tabel 4.2.Coduri de eroare și parametrii rutinelor MPI | 44 |
| Tabel 4.3.Moduri de comunicare MPI | 44 |
| Tabel 5.1.Componente CUDA și semnificația lor | 56 |
| Tabel 10.1.Tipuri de exchangeri | 123 |
| Tabel 11.1.Transformări Spark | 136 |
| Tabel 11.2.Operatori Spark pe seturi de date | 137 |
| Tabel 11.3.Acțiuni Spark | 138 |
| Tabel 11.4.Execuția algoritmilor în diferite contexte de infrastructură | 145 |

Lista Figuri

| | |
|---|-----|
| Figura 1.1.Stările unui thread Java | 4 |
| Figura 3.1.Modelul fork-join OpenMP | 27 |
| Figura 3.2.Tipuri de construcții OpenMP | 28 |
| Figura 5.1.Stiva software CUDA | 54 |
| Figura 5.2.Flux de procesare CUDA | 55 |
| Figura 5.3.Structuri de threaduri | 58 |
| Figura 5.4.Modelul de memorie | 58 |
| Figura 5.5.Modelul hardware | 59 |
| Figura 6.1.Mecanismul socket TCP | 73 |
| Figura 6.2.Mecanismul socket datagrame | 75 |
| Figura 7.1.Arhitectura RMI | 80 |
| Figura 7.2.Scenariul utilizării RMI | 83 |
| Figura 8.1.Ordonarea evenimentelor | 95 |
| Figura 8.2.Exemplu de utilizare a ceasurilor Lamport | 96 |
| Figura 8.3.Exemplu al lipsei de cauzalitate cu ceasuri Lamport | 96 |
| Figura 8.4.Exemplu ceasuri vectoriale | 97 |
| Figura 9.1.Modelele de mesagerie furnizate de JMS API | 108 |
| Figura 9.2.Interfețe generice JMS API | 110 |
| Figura 9.3.Interfețe JMS pentru modelul de comunicare bazat pe cozi de mesaje | 111 |
| Figura 9.4.Interfețe JMS pentru comunicarea în modelul pub/sub | 111 |
| Figura 10.1.Configurație Uni-binding | 128 |
| Figura 10.2.Configurație Multi-binding | 128 |
| Figura 10.3.Configurația aplicației de logging | 128 |
| Figura 10.4.Configurație aplicație ce consumă selectiv mesaje(topicuri) | 129 |
| Figura 11.1.Componentele Spark | 131 |
| Figura 11.2.Cluster Spark | 132 |
| Figura 11.3.Transformări Narrow vs Wide | 134 |
| Figura 11.4.Arhitectura Spark Streaming | 135 |
| Figura 11.5.Reprezentarea partiționării DStream-urilor în RDD | 139 |
| Figura 11.8. Exemplu page rank | 140 |
| Figura 11.9. Modelul de execuție PageRank | 141 |
| Figura 11.10.Transformarea flatMap | 141 |
| Figura 11.11.Transformarea reduceByKey | 142 |