# Algoritmi distribuiți pentru coordonare și sincronizare

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 sau/şi coordoneze acţiunile, motiv pentru care la nivelul unui sistem distribuit apare o coordonată suplimentară și anume timpul. Sincronizarea şi coordonarea joacă un rol foarte important în majoritatea sistemelor distribuite.

Prin *coordonare* se face referire la corelarea acțiunilor proceselor separate și asigurarea unor *mecanisme de comunicare* la nivel global. *Sincronizarea* este similară coordonării însă integrează și factorul timp, referind ordinea evenimentelor şi execuţia instrucţiunilor *raportată la timp*. Un exemplu de coordonare poate fi considerat agreementul (consensul) la care procesele ajung în momentul identificării și alegerii acţiunilor ce urmează a fi executate (de către unul dintre ele sau împreună), iar un exemplu de sincronizare, îl constituie ordonarea evenimentelor distribuite ce ajung într-un fişier de tip log-file şi certitudinea că un proces realizează o anumită acţiune la moment de timp necesar.

### 8.1.Obiective

- Studiul modelelor 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 acelaşi *timp global*. Pe de altă parte, într-un sistem distribuit, fiecare calculator are propriul său semnal de ceas. Datorită faptului că nici un ceas nu poate fi perfect, fiecare dintre aceste semnale au propriile lor întârzieri (aşa numitele *clock skew- variația ceasurilor)* ce referă faptul că semnalul unui ceas ajunge la diferitele componente, în diferite momente de timp. Aceste întârzieri au ca efect variații uneori nepermise pentru ceasurile diverselor sisteme de calcul și astfel acestea pot ajunge în stare desincronizată.

Există diverse concepte referitoare la timp, cu relevanţă într-un sistem distribuit. În primul rând, un ceas intern ţine evidenţe care pot fi translatate în timp fizic (ore, minute, secunde, etc.). Acest timp fizic poate fi global sau local. Timpul global este un timp universal, acelaşi pentru toate componentele, şi este bazat în general pe o formă de timp absolut (timpul global cel mai precis este UTC (Coordinated Universal Time). Pe lângă timpul global, procesele pot avea şi un timp local, caz în care timpul este relevant doar proceselor în mod individual. Timpul local poate să fie bazat pe *ceasuri fizice sau logice*, categorii ce vor fi detaliate în continuare.

În cadrul algoritmilor distribuiţi, în general, sunt utilizate două tipuri de modele de timp și anume un model sincron, timpul necesar realizării tuturor sarcinilor, întârzierile din timpul comunicării şi diferenţele între ceasul fiecărui nod, sunt cu valori limitate, cunoscute, respectiv modelul asincron fără limite de timp. Sistemele distribuite cele mai apropiate de realitate sunt asincrone, însă este mult mai simplă dezvoltarea unui algoritm distribuit pentru un sistem sincron.

#### 8.2.1.Ceasuri fizice și ceasuri logice în sisteme distribuite

Semnalele de ceas fizice ţin evidenţa timpului fizic. În sistemele distribuite care se bazează pe timpul efectiv, este necesară menţinerea semnalului de ceas al fiecărui calculator, în stare sincronizată. Ele pot fi sincronizate cu timpul global (sincronizare externă), sau unele cu altele (sincronizare internă).

*Algoritmul Cristian (Cristian`s Algorithm)* şi Network Time Protocol (NTP) sunt exemple de algoritmi dezvoltaţi pentru sincronizarea semnalelor cu o sursă de timp externă şi totodată globală (de obicei UTC). *Algoritmul Berkeley (The Berkeley Algorithm)* este un exemplu de algoritm care permite sincronizarea semnalelor de ceas în mod intern.

Pentru multe aplicaţii, *ordonarea relativă* a evenimentelor este mult mai importantă decât timpul fizic efectiv. Într-un singur proces, ordonarea evenimentelor (de exemplu, schimbările de stare) este trivială. Într-un sistem distribuit, pe lângă ordonarea locală a evenimentelor, toate procesele trebuie să ajungă la consens cu privire la ordonarea evenimentelor asociate cauzal (de exemplu, trimiterea şi primirea unui mesaj singular).
Fiind dat un sistem ce conţine *N procese* $p_i$, *i luând valori între {1,...,N},* se defineşte o ordonare locală a evenimentelor ( notată $\rightarrow$) ca o relaţie binară, astfel încât, dacă $p_i$ observă evenimentul $e$ înainte de $e'$, astfel avem $e \rightarrow_i e'$. Bazat pe această ordonare locală, definim o *ordonare globală* ca o relaţie de tipul *happened before* $\rightarrow$, definiţia dată de Lamport fiind următoarea:

Relaţia $\rightarrow$ este astfel încât:

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

Relaţia $\rightarrow$ este asimilată unei relaţii de ordonare parţială (îi lipseşte reflexivitatea). Dacă a $\rightarrow$ b, atunci spunem că *a îl afectează cauzal pe b*. Considerăm că evenimentele neordonate sunt *concurente*.

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

$E11 \rightarrow E12, E13, E14, E23, E24, ...$
$E21 \rightarrow E22, E23, E24, E13, E14, ...$

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

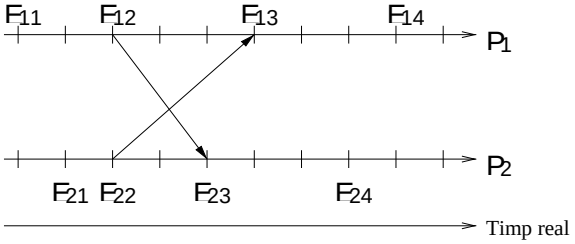


**Figura 8.1. Ordonarea evenimentelor**

### 8.2.2.Ceasuri logice Lamport

Ceasurile logice Lamport pot fi implementate sub forma unor numărătoare care calculează local relaţia de tipul *happened-before* $\rightarrow$. Aceasta înseamnă că, fiecare proces $p_i$ *reţine un ceas logic* $L_i$. Fiind dat un asemenea semnal, $L_i(e)$ denotă un timestamp de tipul Lamport al evenimentului $e$ la $p_i$ şi $Lj(e)$ 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 local, un proces $p_i$ execută $L_i := L_i + 1$.
2. De fiecare dată când un mesaj m este trimis de la $p_i$ la $p_j$:

- Procesul $p_i$ execută $L_i := L_i + 1$ şi trimite noul $L_i$ *cu m*.
- Procesul $p_j$ recepţionează $L_i$ *cu m* şi execută $L_j := \max(L_j, L_i) + 1$. *receive*($m$) este notat cu noul $L_j$.

În această schemă, $a \rightarrow b$ implică $L(a) < L(b)$, dar $L(a) < L(b)$ nu implică în mod necesar $a \rightarrow b$. Ca un exemplu, considerăm Figura 8.2. În această figură $E_{12} \rightarrow E_{23}$ şi $L_1(E_{12}) < L_2(E_{23})$ , totodată avem şi $E_{13} \rightarrow E_{24}$ cât timp $L_1(E_{13}) < L_2(E_{24})$ .

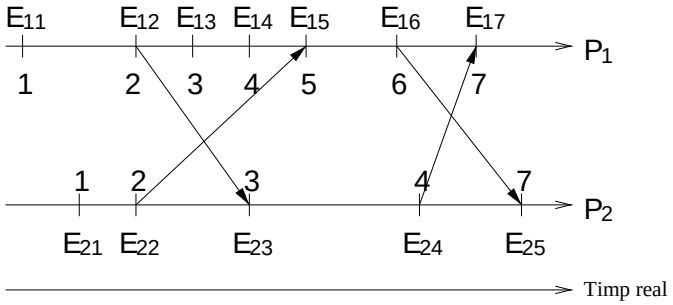


**Figura 8.2. Exemplu de utilizare a ceasurilor Lamport**

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

### 8.2.3.Ceasuri logice vectoriale

Principala deficienţă a ceasurilor logice Lamport este faptul că $L(a) < L(b)$ nu implică $a \rightarrow b$; prin urmare, nu putem deduce dependenţe cauzale din timestamps. De exemplu, în Figura 8.3, avem $L_1(E_{11}) < L_3(E_{33})$, dar $E_{11}$ nu implică $E_{33}$.
Esența problemei o constituie înaintarea independentă sau prin mesaje a ceasurilor ,fără menținerea unui istoric al acestei înaintări.Problema poate fi rezolvată prin trecerea de la ceasuri scalar,e la vectori de ceasuri, unde fiecare proces reţine un vector $V_i$.

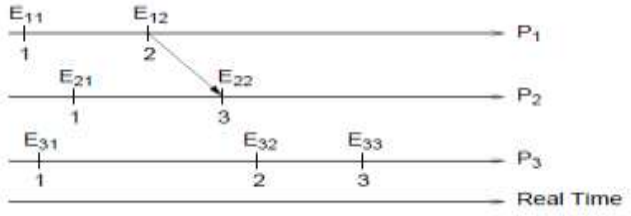


**Figura 8.3. Exemplu al lipsei de cauzalitate cu ceasuri Lamport**

$V_i$ *este un vector de mărime N,* unde N este numărul de procese. Componenta $V_i[j]$ conţine informaţiile pe care le deţine procesul $p_i$ *referitoare la mesajul* $p_j$. Iniţial, avem $V_i[j] := 0$ pentru $i,j \in \{1,...,N\}$.

Mesajele înaintează astfel:

1. Înainte ca $p_i$ să atribuie un timestamp unui eveniment, execută $V_i[i] := V_i[i] + 1$.
2. De fiecare dată când un mesaj m este trimis de la $p_i$ la $p_j$:
   Procesul $p_i$ execută $V_i[i] := V_i[i] + 1$ şi trimite $V_i$ cu *m*.
   Procesul $p_j$ primeşte $V_i$ cu *m şi* adună semnalele de vector $V_i$ şi $V_j$ astfel:
   $Vj[k]$:=max($Vj[k],V_i[k]$) + 1 *,dacă j = k*
   $Vj[k]$:= max($V_j[k],V_i[k]$) , *dacă j # k*

Astfel ,se asigură faptul că tot ceea ce se întâmplă ulterior la procesul $p_j$ , este acum în relaţie cauzală cu tot ceea ce s-a întâmplat la $p_j$.În această abordare , avem pentru oricare $i,j$, $V_i[i] \geq V_j[i]$ (de

exemplu, $p_i$ are întotdeauna cea mai nouă versiune a propriului ceas); mai mult, $a \rightarrow b$ *dacă şi numai dacă* $V(a) < V(b)$.

De exemplu, considerăm notaţiile din diagrama reprezentată în Figura 8.4. Fiecare eveniment este notat atât cu valoarea vectorului de ceas propriu (tripletul), cât şi valoarea corespondentă a unui ceas scalar Lamport.

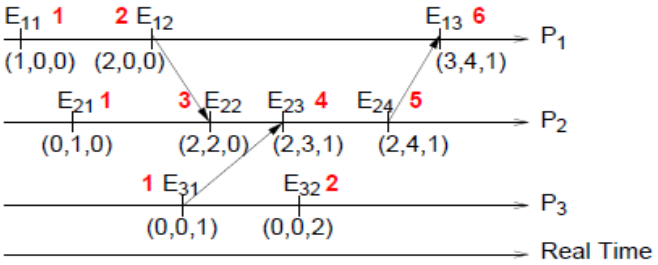


**Figura 8.4. Exemplu ceasuri vectoriale**

### 8.2.4.Algoritmi pentru ceasuri logice

Pentru cei doi algoritmi descriși în continuare , 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 sincron de mesaje . 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

Informații suplimentare ceasuri vectoriale https://en.wikipedia.org/wiki/Vector_clock

**Algoritmul 1. Birman-Schiper-Stephenson**

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

Resurse
https://www.geeksforgeeks.org/birman-schiper-stephenson-protocol/
https://github.com/besuikerd/distributedalgorithms/tree/master/BirmanSchiperStephenson
https://github.com/adimitrova/DistributedAlgorithms

*Pseudocodul protocolului*
adaptat din
http://nob.cs.ucdavis.edu/classes/ecs251-2000-01/distclock.html

Notație

*n* procese

$P_i$ proces

$C_i$ ceas vectorial asociat procesului $P_i$; elementul *j* este $C_i[j]$ și conține pentru $P_i$ ultima valoare pentru timpul curent în procesul $P_j$

$t^m$ timestamp vectorial pentru mesajul *m* (după incrementarea ceasului local)

**$P_i$ trimite un mesaj către $P_j$**

1. $P_i$ incrementează $C_i[i]$ și setează timestamp $t^m = C_i[i]$ pentru mesajul *m*.

**$P_j$ recepționează un mesaj de la $P_i$**

1. Când $P_j$, $j$ != $i$, recepționează *m* cu timestampul $t^m$, va întârizia livrarea mesajului până când ambele condiții sunt îndeplinite:
   a. $C_j[i] = t^m[i] - 1$; și
   b. pentru $k <= n$ și $k$ != $i$, $C_j[k] <= t^m[k]$.
2. La livrarea mesajului către $P_j$, se actualizează ceasul vectorial al lui $P_j$.
3. Verifică mesajele buferate pentru a vedea dacă sunt mesaje ce pot fi livrate.

## Algoritmul 2. Schiper-Eggli-Sandoz Protocol

Scop: asigură că mesajele vor ajunge la procesul receptor, în ordinea trimiterii. Acest protocol nu va folosi mesaje broadcast. Fiecare mesaj are asociat un vector ce conține informație pentru fiecare receptor astfel încât acesta să determine corect dacă un alt mesaj l-a precedat. Ceasurile se actualizează doar la trimiterea mesajelor.

Resurse
https://link.springer.com/chapter/10.1007/3-540-51687-5_45
https://github.com/dzzh/in4150/tree/master/1
https://github.com/johannmeyer/Schiper-Eggli-Sandoz

### *Pseudocodul protocolului*

*adaptat din*
*http://nob.cs.ucdavis.edu/classes/ecs251-2000-01/distclock.html*

Notație

*n* procese

$P_i$ proces

$C_i$ ceas vectorial asociat procesului $P_i$; elementul j este $C_i[j]$ și contine în $P_i$ ultima valoare pentru timpul curent în procesul $P_k$

$t^m$ timestamp vectorial pentru mesajul *m* (stampă după incrementarea ceasului local)

$t^i$ timp curent la procesul $P_i$

$V_i$ vector pentru procesul $P_i$ de forma $V_i[j] = t^m$, unde $P_j$ este procesul destinație și $t^m$ stampa de timp vectorială; $V_i[j][k]$ este componenta k din $V_i[j]$. $V^m$ vectorul corespunzător mesajului *m*

**$P_i$ trimite un mesaj către $P_j$**

1. P*i* trimite mesajul *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 $P_j$, $j$ != $i$, primește *m*, va întârzia livrarea mesajului dacă ambele condiții sunt îndeplinite:
   a. $V^m[j]$ este setat; și
   b. $V^m[j] < t^j$

2. Atunci când mesajul este livrat către $P_j$, se vor actualiza 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 , nu se realizează nimic.
    b. dacă $V_j[k]$ este neinițializat și $V^m[k]$ este inițializat, se setează $V_j[k] = V^m[k]$.
    c. dacă ambele $V_j[k]$ și $V^m[k]$ sunt inițializate, se setează $V_j[k][k'] = \max(V_j[k][k'], V^m[k][k'])$ pentru $k' = 1, ..., n$
3. Actualizează ceasul vectorial al lui $P_j$ .
4. Verifică mesajele buferate pentru a vedea dacă pot fi livrate

### 8.2.5.Algoritmi pentru alegere leader

Diverse procesări în context distribuit necesită un anumit proces cu rol distinctiv (lider sau coordonator) determinat din cadrul unui set de procese, astfel încât procesarea să poată fi coordonată de acesta. În prezența eșecului procesului ales drept lider, este necesară determinarea unui nou lider pentru a permite continuarea procesării. Pentru cazul în care toate procesele au un număr de identificare unic, alegerea liderului poate fi redusă la operaţia de găsire a unui proces neeşuat, ce deține cel mai mare identificator.

Informații suplimentare conceptualizare algoritmi :
https://en.wikipedia.org/wiki/Leaderelection
https://docs.microsoft.com/en-us/azure/architecture/patterns/leader-election

Orice algoritm corect ,de determinare a acestui proces trebuie să îndeplinească următoarele proprietăţi:
1. Siguranţă: un proces, fie nu cunoaşte coordonatorul, fie cunoaşte identificatorul procesului cu cel mai mare număr de identificare.
2. Longevitate: în cele din urmă, un proces este supus eşecului sau îl cunoaşte pe coordinator.

Unul din cei mai cunoscuți algoritmi de alegere lider a fost propus de Garcia-Molina (Bully algorithm) şi utilizează trei tipuri de mesaje:
1. Mesaje de tipul Election (alegere) – anunţă alegerea
2. Mesaje de tipul Answer (răspuns) – procesele răspund la o alegere
3. Mesaje de tipul Leader (coordonator) – coordonatorii aleşi se anunţă în sistem

**Algoritmul Bully.** Modul de execuție al 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 eşuat, sau dacă primeşte un mesaj de tipul Election. Când se începe procesul de alegere, un proces trimite mesajul Election la toate procesele cu un număr de identificare mai mare. Dacă procesul nu primeşte niciun mesaj de tipul Answer într-un interval de timp predefinit, procesul care a început alegerea va decide că trebuie să devină coordonator şi va trimite un mesaj de tipul Coordinator către toate celelalte procese.

În cazul în care primeşte un mesaj de tipul Answer, procesul care a iniţiat alegerea va aştepta un interval de timp predefinit pentru un mesaj de tipul Coordinator. Un proces care primeşte un mesaj de tipul Election poate anunţa instantaneu că el este coordonatorul dacă ştie sigur că deţine numărul de identificare cel mai mare. Altfel, va începe şi el o alegere subordonată prin trimiterea unui mesaj de tipul Election către procesele care au numerele de identificare mai mari. Algoritmul este numit și algoritm bully (forța brută) deoarece procesul cu numărul cel mai mare de identificare va fi mereu coordonator.

*Algoritm Bully* (Garcia-Molina ,1982) adaptare din http://www.cs.vu.nl/~tcs/da/daslides.pdf
Resurse https://github.com/brijeshbhalodiya/bully-algorithm-java/tree/master/src/main/java

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**
   deoarece 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 un algoritm cu organizare topologică a proceselor de tip inel. În această abordare, toate procesele sunt ordonate sub forma unui inel logic şi fiecare proces cunoaşte informaţii despre structura inelului. Mesajele implicate sunt doar de două tipuri – mesaje de tipul *Election* şi de tipul *Lider*. Un proces începe o alegere în momentul în care realizează că precedentul coordonator a eşuat. O nouă alegere porneşte prin trimiterea unui mesaj de tipul Election la primul vecin de pe inel. Mesajul conţine identificatorul procesului, acesta este transmis prin inel, fiecare proces adăugându-şi propriul identificator. Alegerea este completă când mesajul ajunge înapoi la procesul iniţiator. Bazat pe conţinutul mesajului, procesul respectiv determină numărul de identificare maxim şi trimite un mesaj de tipul *Lider*, specificând faptul că procesul respectiv este câştigătorul.

*Algoritm Ring* (Chang Roberts 1979), adaptare din http://www.cs.vu.nl/~tcs/da/daslides.pdf
Resurse :https://github.com/Rabshab/ChangRoberts

1. Procesele sunt aranjate într-un inel în ordinea crescătoare a numărului procesului
2. Când un proces constată că liderul este căzut, procesul trimite un mesaj "election" la următorul proces din ring
   a. Mesajul "election" conține o listă cu toate numerele de proces care au văzut până acum mesajul "election"
   b. Inițial, lista conține numărul procesului care a inițiat alegerile
3. Procesul așteaptă să primească o confirmare de la destinatar
4. Dacă nu există nicio confirmare, procesul trimite mesajul "election" la următorul proces în ring și așa mai departe până când primește o confirmare
5. Când un proces primește un mesaj "election", procesul se adaugă la lista proceselor din mesaj și transmite mesajul mai departe
6. În cele din urmă procesul care a început alegerile primește un mesaj "election" care a parcurs întreg ringul
   a. Lista din mesaj conține toate procesele care sunt încă în desfășurare
   b. Procesul cel mai mare număr va fi liderul
   c. Procesul schimbă mesajul într-un mesaj "leader" și îl transmite în jurul inelului
7. Când un proces primește un mesaj "leader", procesul înregistrează cine este noul lider și transmite mesajul

8. În cele din urmă, procesul care a început alegerile primește un mesaj "lider" care a parcurs întreg ringul

a. Procesul nu mai transmite mesajul și alegerile se termină

### 8.2.6.Excluderea mutuală distribuită

Unele dintre problemele întâmpinate în domeniul concurenţei din cadrul sistemelor distribuite sunt similare cu cele întâlnite în studiul sistemelor de operare şi a aplicaţiilor multithread. În particular, ne referim aici la problemele impuse de condiţiile de concurenţă care apar în momentul în care procese multiple accesează resurse comune. În sistemele nedistribuite, aceste probleme sunt rezolvate prin implementarea excluderii mutuale, implicând folosirea primitivelor sistemului de operare (primitive locale) cum ar fi lacătele, semafoarele şi monitoarele. În sistemele distribuite, rezolvarea problemelor datorate concurenţei devine mult mai complexă, datorită lipsei de resurse comune directe (cum ar fi memoria, regiştrii CPU, etc.), datorită lipsei unui ceas global, a unei stări globale unice şi datorită prezenţei întârzierilor de comunicare.

Atunci când se dorește accesul concurent la o resursă dintr-un sistem distribuit, e necesară existenţa unor mecanisme care ajută la prevenirea condiţiilor de concurenţă în timp ce procesele se află în cadrul unor secţiuni critice. Aceste mecanisme trebuie să asigure existenţa următoarelor trei proprietăţi: a)siguranţa: cel mult un proces are dreptul de a executa secţiunea critică la un moment dat. b) longevitatea: la un moment dat, cererile de a intra şi de a ieşi din secţiunea critică trebuie să fie efectuate cu succes și c) ordonarea: cererile sunt procesate într-o ordine de tipul happened-before.

Au fost propuse mai multe soluții pentru rezolvarea problemei accesului concurrent la resurse în cazul proceselor distribuite și anume : metoda unui Server central, metoda Token Ring sau algoritmi 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 şi ieşirea din secţiunile critice. Procesele trebuie să trimită cereri de intrare şi ieşire dintr-o secţiune critică către un server coordonator. Acesta acordă permisiuni de intrare prin trimiterea unui *token* spre procesul care a înaintat cererea. La părăsirea secţiunii critice, token-ul este returnat serverului. Procesele care doresc să intre într-o secţiune critică în timp ce un alt proces deţine token-ul, sunt aşezate într-o coadă. Când token-ul este returnat, procesul de la începutul cozii primeşte token-ul şi totodată permisiunea de a intra în secţiunea critică.Această schemă este uşor de implementat, dar nu prezintă o scalabilitate bună, datorită existenţei unei autorităţi centrale , mai mult, este vulnerabilă din punctul de vedere al eşuării serverului central.

**Algoritm 2: Token Ring (inel cu jeton)** .O topologie de procese mai sofisticată este cea care organizează toate procesele într-o structură logică de inel, alături de un mesaj de tipul token care parcurge inelul continuu. Înainte de a intra în secţiunea critică, un proces trebuie să aştepte până când token-ul trece prin dreptul său. În acel moment, procesul reţine token-ul până în momentul părăsirii secţiunii critice. Dezavantajul acestei abordări este faptul că inelul implică o întârziere de N/2 salturi, lucru care limitează scalabilitatea din nou. Mai mult, mesajele de tipul token consumă lăţime de bandă şi nodurile sau canalele eşuate pot rupe inelul. O altă problemă este reprezentată de pierderea token-ului – fapt cauzat de prezenţa eşecurilor. În plus, dacă un proces nou se alătură reţelei sau doreşte să o părăsească, este nevoie de logică adiţională de management pentru integrare.

*Comparație și analiza algoritmilor*

Pentru compararea algoritmilor de excludere mutuală distribuită este necesară analiza numărului de mesaje inter-schimbate per intrare/ieşire dintr-o secţiune critică, întârzierea unui proces înainte ca acesta că primească permisiunea de a intra în secţiunea critică, respectiv nivelul de siguranță al algoritmilor (ce probleme posibile întâmpină aceștia în execuție).

Algoritmul centralizat are nevoie de un total de 3 mesaje inter-schimbate la fiecare execuţie a unei secţiuni critice (două pentru intrare şi unul pentru ieşire). După ce un proces trimite cererea pentru obţinerea permisiunii de a intra într-o secţiune critică, acesta trebuie să aştepe pentru un timp minim în care are loc inter-schimbarea a două mesaje (unul pentru deţinătorul curent - ca să returneze token-ul coordonatorului, şi unul pentru coordinator – ca să trimită token-ul la procesul aflat în aşteptare). Cea mai mare problemă pe care o are acest algoritm referă situația în care coordonatorul eşuează (sau devine nedisponibil) şi prin urmare, tot algoritmul eşuează.

Pentru algoritmul inel, numărul mesajelor inter-schimbate per intrare şi ieşire din secţiunea critică, depinde de cât de des au nevoie procesele să intre în secţiune. Cu cât mai rar doresc procesele să intre, cu atât mai mult timp va dura drumul token-ului prin inel şi cu atât mai mari vor fi costurile ,în termenii mesajelor inter-schimbate de intare într-o secţiune critică.

Din punctul de vedere al întârzierilor depinzând de locaţia token-ului, va exista un număr de 0 până la n-1 mesaje ,înainte ca un proces să intre în secţiune. Cele mai mari probleme întâmpinate de acest algoritm sunt pierderea tokenului şi ruperea inelului de către un proces eşuat. Este posibilă, desigur, prevenirea acestora, prin oferirea tuturor proceselor de informaţii referitoare la structura inelului. În acest mod, nodurile eşuate pot fi evitate.

**B)Algoritmi de excludere mutuală ce folosesc ceasuri logice**

**Algoritm 1. Algoritmul Ricart-Agrawala**

Ricart şi Agrawala [Ricart- Agrawala 1981] au propus un algoritm pentru excluderea mutuală distribuită, care utilizează ceasuri logice. fiecare proces participant $p_i$ reţine un ceas Lamport şi toate procesele trebuie să fie capabile să comunice în perechi. În orice moment, un proces se poate afla doar în una din următoarele trei stări:

1. *Released*: În afara secţiunii critice
2. *Wanted*: Aşteptând să intre în secţiunea critică
3. *Held:* În interiorul secţiunii critice

Dacă un proces doreşte să intre în secţiunea critică, realizează operaţia de multicast a unui mesaj $L_i$,$p_i$ şi aşteaptă până la primirea unui răspuns de la oricare alt proces. Procesele operează astfel:

- Dacă un proces este în starea *Released,* va răspunde instantaneu la oricare cerere de intrare în secţiunea critică.
- Dacă procesul este în starea *Held,* va întârzia răspunsul la oricare cerere până în momentul ieşirii din secţiunea critică.
- Dacă procesul este în starea *Wanted*, va răspunde la oricare cerere instantaneu doar dacă timestamp-ul cererii este mai mic decât cel din cadrul propriei sale cereri.

Caracteristicile algoritmului sunt:

- Imbunătățirea algoritmului original propus de Lamport
- Control distribuit
- Necesită o metodă de asignare a timpilor către evenimente
- Nu există ceas global
- folosește „ceasul Lamport”
- Fiecare proces/procesor își păstreaza propriul timp
- Evenimentele locale sunt asignate în ordinea strictă a creșterii marcajelor de timp
- Fiecare mesaj dintre procese e însoțit de un marcaj de timp care indică timpul procesului care a trimis mesajul

- Când un mesaj este recepționat, marcajul său de timp este comparat cu timpul local
- Daca timestamp > timp local, setează timp local = timestamp + 1

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

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

Această ordonare nu e o ordonare totală, dar poate fi transformată într-una totală,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://www.geeksforgeeks.org/ricart-agrawala-algorithm-in-mutual-exclusion-in-distributed-system/
https://github.com/vineetdhanawat/ricart-agrawala
https://github.com/madhav5589/Ricart-Agrawala-algorithm-for-distributed-mutual-exclusion

*Pseudocodul algoritmului (https://www.cs.uic.edu/~ajayk/Chapter9.pdf):*

```
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 aceasa și ori va elimina mesajul din grup, ori va opri aplicația. Variante mult mai scalabile ale acestui algoritm cer fiecărui proces individual să contacteze subseturi ale vecinilor în momentul în care se dorește intrarea în secţiunea critică. Din păcate, eşecul oricărei componente -proces poate afecta intrarea oricăror altor procese în secţiunea critică deci poate periclita execuția algoritmului.

**Algoritm 2 .Algoritmul Maekawa**

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

- Intersectia lui Ri cu Rj nu trebuie să fie vidă
- Fiecare proces aparține mulțimii sale de cereri
- Fiecare mulțime de cereri are același număr de elemente K
- Fiecare proces aparține la exact K mulțimi de cereri

Maekawa a arătat că este posibilă construirea unei mulțimi astfel încât K = O(radical(N)). Algoritmul foloseste trei tipuri de mesaje: *Request, ok, End*. Un proces care solicită intrarea într-o zonă critică trebuie să primeasca OK-uri doar de la membri mulțimii sale de solicitanți.

Informații suplimentare :
https://en.wikipedia.org/wiki/Maekawa's_algorithm
https://www.geeksforgeeks.org/maekawas-algorithm-for-mutual-exclusion-in-distributed-system/
https://github.com/saransh2405/Maekawa-algorithm,https://github.com/sanketc/MaekawaAlgorithm

*Pseudocodul algoritmului (https://www.cs.uic.edu/~ajayk/Chapter9.pdf):*

```
Enter CS:

Send request message to each member of my request set.
Wait for okay messages from each member of my request set.

Exit CS:

Send done message to ecah member of my request set.
receiveRequestMessage
( each process has a granted variable, initialized to false)
 If granted
enqueue the request;
else {granted =true;
   send okay to the requestor;}
 receiveDoneMessage:
if (queue is empty)
granted=false ;else{
   dequeue the request with the earliest timestamp;
send okay top the requestor;}
```

Avantajul acestei metode este faptul ca doar 3*radical(N) mesaje sunt necesare pentru intrarea intr-o zona critică.Problema posibilă este de aparitie a unui blocaj (deadlock) la nivelul proceselor, iar 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 inițial poate pune apoi solicitarea primită înapoi în coada de așteptare și trimite un mesaj de *ok* către solicitarea pe care tocmai a primit-o

*Comparație și analiza algoritmilor*

Algoritmul original al lui Lamport presupune ca fiecare proces sa solicite acces de la celelalte procese, folosind mesaje de 3 tipuri: request, reply, release. Aceasta determină ca pentru fiecare solicitare către zona critica (critical section, presurtat CS), sa fie nevoie de 3*(N-1) mesaje, unde N este numarul de noduri/procese.

In comparație cu soluția originală Lamport, algoritmul Ricart-Agrawala vine cu o reducere de mesaje transmise la 2*(N-1), deoarece folosește mesaje doar de tip request și reply. Mai mult de atât, algoritmul lui Maekawa scade numărul de mesaje la 3*radical(N), deoarece el are 3 tipuri de mesaje (request, locked, release), dar ele sunt transmise numai în 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, folosește o coadă fifo uzuală.

### 8.3.Exemple

8.3.1.Se vor testa exemplele ce ilustrează algoritmii distribuiți prezentați în lucrare , ce pot fi descărcați de la adresa https:/ftp.utcluj.ro /~civan/CPD/2_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 avantaje, dezavantaje, posibile optimizări și număr de mesaje schimbate.

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

### 8.5.Probleme propuse

8.5.1.Trasați execuția algoritmilor distribuiți utilizând la alegere unul din simulatoarele ce pot fi descărcate de la adresele

https://www.weizmann.ac.il/sci-tea/benari/software-and-learning-materials/daj-distributed-algorithms-java, sau

http://iao.hfuu.edu.cn/news/32-programming-blog/234-distributed-algorithms-simulator

8.5.2.Implementați algoritmii de alegere lider (Bully și Ring) realizați și prezentați o analiză a complexității lor ( mesaje schimbate, timp de execuție, scalabilitate). Identificați posibile scenarii de aplicabilitate a fiecărui algoritm.

8.5.3.Implementați algoritmii de excludere mutuală distribuită *clasici* (Server central și Token Ring) prezentați și realizați o analiză a complexității lor (mesaje schimbate, timp de execuție , scalabilitate).Identificați posibile scenarii de aplicabilitate a fiecărui algoritm.

8.5.4.Implementați algoritmii de excludere mutuală bazați pe *ceasuri logice* (Ricart Agrawala și Maekawa) prezentați și realizați o analiză a complexității lor. Identificați scenarii adecvate de aplicabilitate a fiecărui algoritm.

## 8.6. Referințe bibliografice

1.W. Fokkink – Distributed algorithms notes : https://www.cs.vu.nl/~tcs/da/daslides.pdf
2. A Kshemkalyani – Distributed Computing-principles, algorithms and systems
https://eclass.uoa.gr/modules/document/file.php/D245/2015/DistrComp.pdf