Sisteme și algoritmi distribuiți
Curs 8

Recapitulare: Ceasuri vectoriale

Fiecare proces P_i stochează vectorul V_i de dimensiune n (inițializat la 0), unde n este numărul de procese

$v_i[i]$ = nr. de evenimente executate pe P_i

$v_i[j]$ = nr. de evenimente de care P_i știe că au fost executate pe P_j

Noua actualizare:

Eveniment local la P_i: $V_i[i] = V_i[i] + 1$

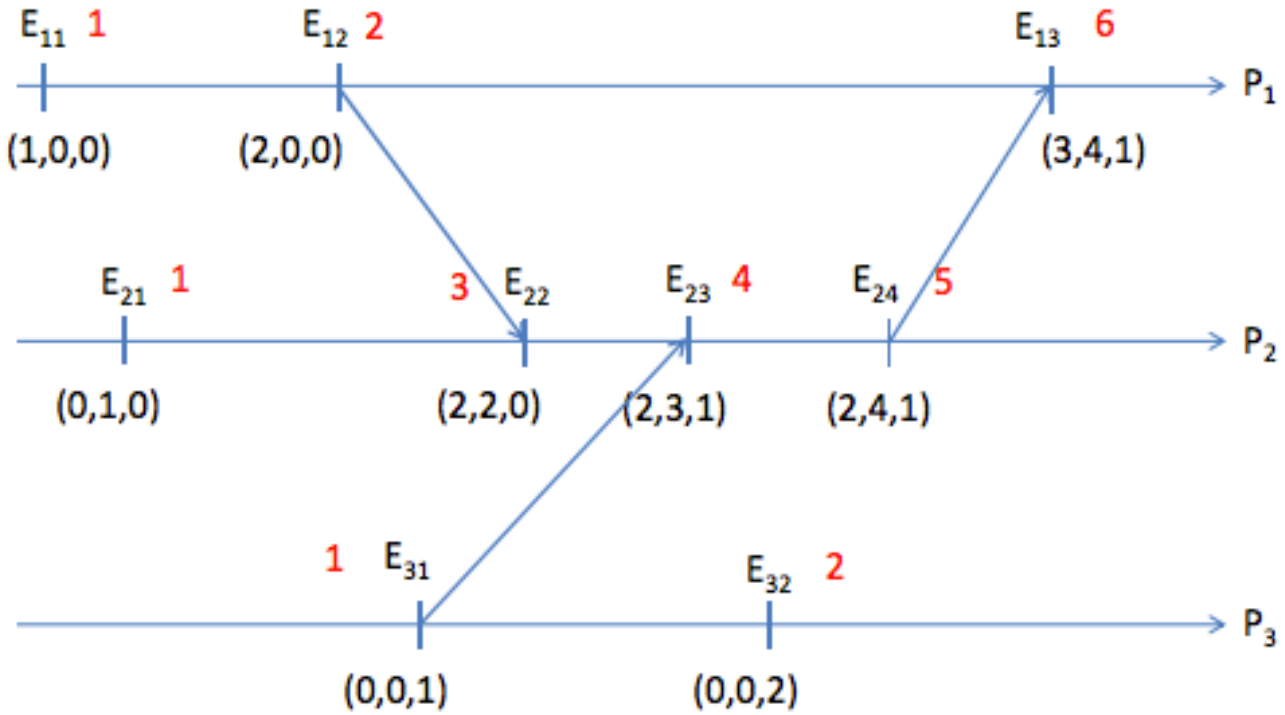
Când m este livrat de P_i la P_j atașează V_i la mesajul m

Recepționează P_j: $V_j[k] = \max\left(V_j[k], V_i[k]\right), \ j \neq k; \ V_j[j] = V_j[j] + 1$

Nodul P_j primește informație despre nr. de evenimente despre care sursa P_i știe că au avut loc la procesul P_k!

Recapitulare: Ceasuri vectoriale

1. Avem $V(A) < V(B)$ dacă și numai dacă A precede cauzal pe B!
2. $V(A) < V(B)$ se definește $V(A) \leq V(B)$ pentru toți i și $\exists\ k\ a.\hat{i}.\ V(A)[k] < V(B)[k]$
3. A și B sunt concurente dacă și numai dacă $V(A)! < V(B)$ și $V(B)! < V(A)$

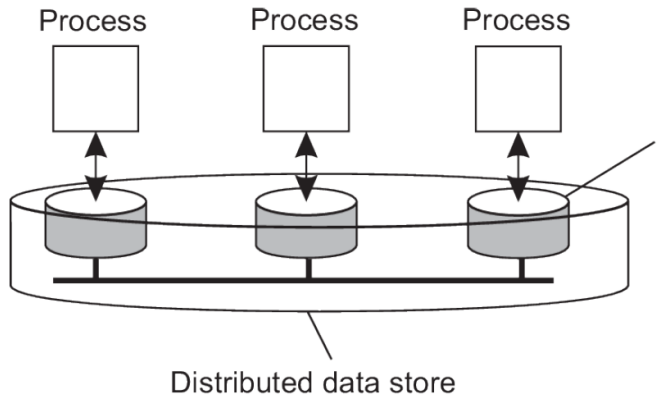


Cuprins

- Sisteme distribuite cu memorie partajată
- Modele de consistență
- Algoritmi
- Excludere mutuală

Consistență și replicare

Sisteme distribuite cu memorie partajată



Sistem format din noduri (sit-uri, procese) care comunică prin intermediul operațiilor de citire-scriere.

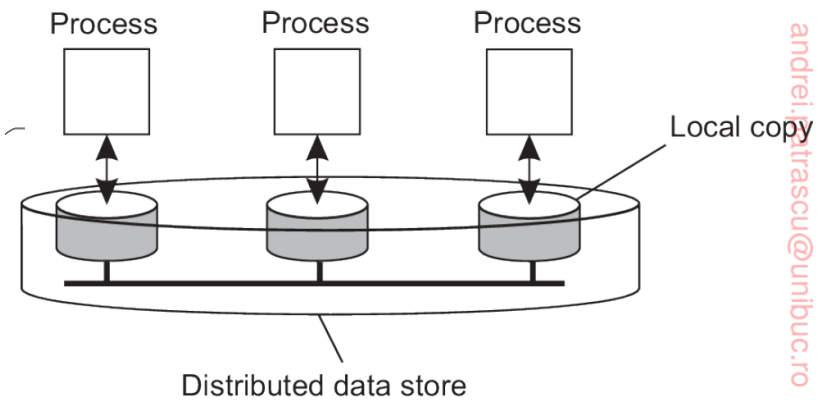
Ipoteze:

- O replică (copie) a memoriei partajate este menținută local de fiecare nod.
- O operație de citire-scriere poate avea loc din oricare nod al sistemului.
- Prin rețeaua de comunicație, operația se propagă în celelalte replici.

Consistența: replicile converg asimptotic către consensul (persistent).

Sisteme distribuite cu memorie partajată (SDMP)

- *Implementare: Data-store* = serviciu de stocare a datelor: baze de date, sisteme de fișiere, severe web.
- Un data-store constă într-un set de noduri-server care conțin copii ale tuturor obiectelor de date
  - poate fi citit - scris de oricare proces din SD
  - O copie locală (replica) poate suporta “citiri rapide”
- Un client se poate conecta la o singură replică
  - Citirile se execută local
  - Scrierile se execută mai întâi local și, după aceea, sunt propagate către celelalte replici.



Modele de consistență

În SDMP poate apărea inconsistența:

- Datelor: un segment de date este *expirat* (*stale*).
- Operațiilor: operațiile sunt executate în ordine diferită pe replici diferite.

Model de consistență = un set de premise pe care procesele din SDMP le respectă cu privire la care combinații de operații sunt admisibile.

Dacă toate nodurile se supun regulilor (protocoale specifice), atunci rezultate de consistență vor fi obținute.

Modele de consistență

Toate nodurile (clienții) care accesează datele vor vedea operațiile într-o ordine conformă cu:

- Consistența strictă
- Consistența secvențială
- Consistența cauzală
- Consistența eventuală

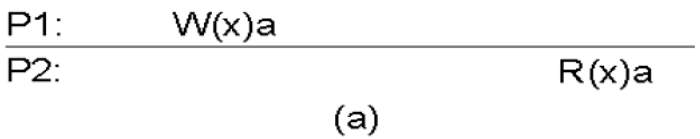
Consistența strictă (linearizabilitate)

Orice eveniment de citire a unui obiect de date returnează rezultatul celui mai recent eveniment de scriere asupra aceluiași obiect de date;

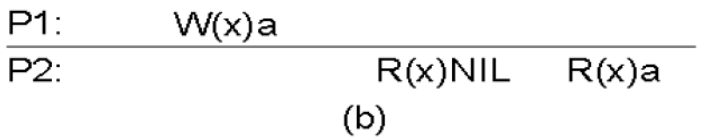
În particular, necesită ca toate nodurile sa dețină:

- *Noțiune de timp global absolut*
- Propagarea instantanee a actualizărilor între replici

Consistența strictă (linearizabilitate)



(a)



(b)

a) Respectă consistența strictă
b) Nu respectă consistența strictă

Imposibil de implementat într-un SDMP real

Consistența secvențială

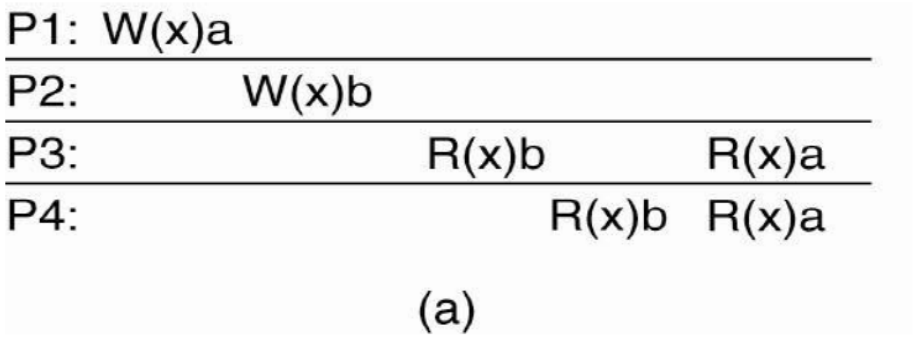
Model de consistență mai relaxat decât consistența strictă.

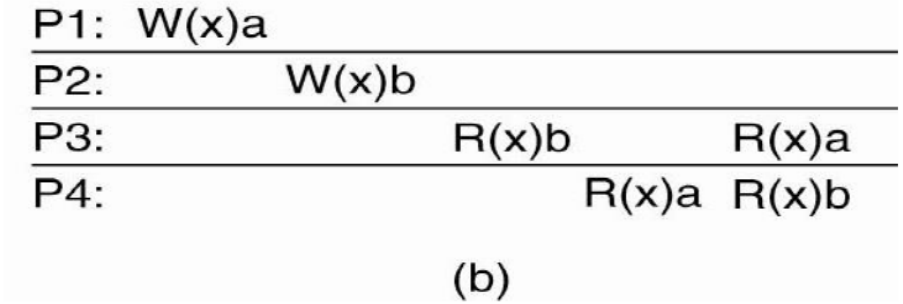
Cerință:

Toți clienții văd operațiile de scriere în aceeași ordine:

- Pp. că toate operațiile sunt executate în ordine secvențială
- Ordinea operațiilor de scriere executate de un singur proces se menține global
- Toate procesele văd aceeași ordine a operațiilor

Consistența secvențială





- În figura (a), P_3 *și* P_4 citesc valoarea b, și după aceea a. (consistența secvențială)
- În figura (b), P_3 *și* P_4 citesc valorile în ordine diferită, echivalent, văd execuția operațiilor de scriere în ordine diferită.

Consistența secvențială

Process P_1	Process P_2	Process P_3
x ← 1;	y ← 1;	z ← 1;
print(y,z);	print(x,z);	print(x,y);

(Lamport, 1979) În termeni de programare distribuită, execuția care respectă consistența secvențială asigură păstrarea ordinii instrucțiunilor din fiecare proces.

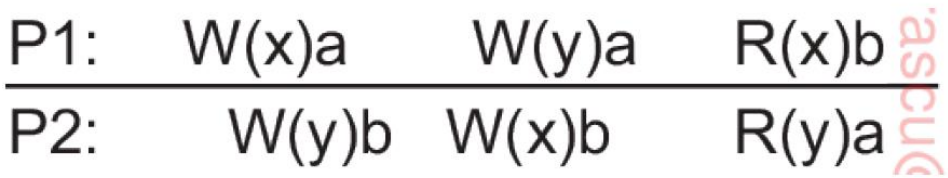
- 720 (6!) posibile execuții
- Din cele care încep cu $x \leftarrow 1$; *(120)* jumătate au print(x, z); înainte de y ← 1; și mai departe, jumătate au print(x, y); înainte de z ← 1. De aceea rămân valide doar 30.
- În total, doar 90 păstrează ordinea locală a instrucțiunilor din fiecare proces.

Consistența secvențială

Process P_1	Process P_2	Process P_3
`x ← 1;`	`y ← 1;`	`z ← 1;`
`print(y,z);`	`print(x,z);`	`print(x,y);`

Execution 1	Execution 2	Execution 3	Execution 4
P_1: `x ← 1;`	P_1: `x ← 1;`	P_2: `y ← 1;`	P_2: `y ← 1;`
P_1: `print(y,z);`	P_2: `y ← 1;`	P_3: `z ← 1;`	P_1: `x ← 1;`
P_2: `y ← 1;`	P_2: `print(x,z);`	P_3: `print(x,y);`	P_3: `z ← 1;`
P_2: `print(x,z);`	P_1: `print(y,z);`	P_2: `print(x,z);`	P_2: `print(x,z);`
P_3: `z ← 1;`	P_3: `z ← 1;`	P_1: `x ← 1;`	P_1: `print(y,z);`
P_3: `print(x,y);`	P_3: `print(x,y);`	P_1: `print(y,z);`	P_3: `print(x,y);`
Prints: `001011`	*Prints:* `101011`	*Prints:* `010111`	*Prints:* `111111`
Signature: `00 10 11`	*Signature:* `10 10 11`	*Signature:* `11 01 01`	*Signature:* `11 11 11`

Consistența secvențială



Pentru aceste operații avem CS

- Dacă urmărim DOAR operațiile asupra variabilei x și schimbăm ultima citire cu $R_1(x)a$, de asemenea obținem un șir de operații secvențial.
- La fel în cazul variabilei y ($R_2(y)b$).
- Cu toate acestea, urmărind perechea (x, y), operațiile de citire $(R_1(x)a, R_2(y)b)$ nu conduc la o execuție consistentă secvențial (neserializabile).

Consistența secvențială

P1:	W(x)a	W(y)a	R(x)b
P2:	W(y)b	W(x)b	R(y)a

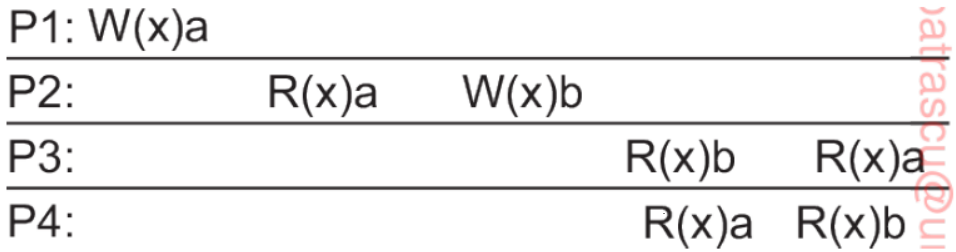
Ordering of operations	Result	
$W_1(x)a; W_1(y)a; W_2(y)b; W_2(x)b$	$R_1(x)b$	$R_2(y)b$
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_2(x)b; W_1(x)a; W_1(y)a$	$R_1(x)a$	$R_2(y)a$

Consistența cauzală

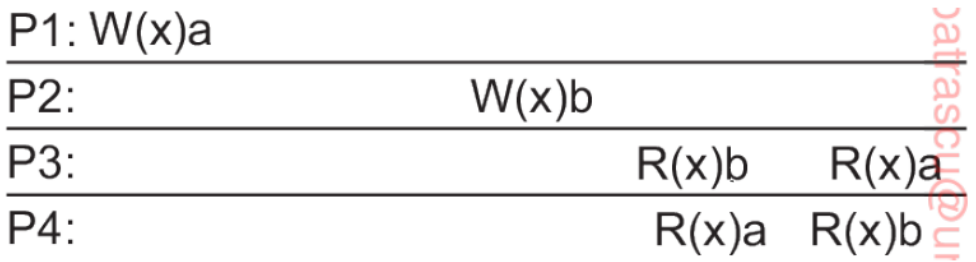
- Relaxează mai departe cerințele consistenței secvențiale.
- Două operații sunt în relație cauzală dacă:
  - o citire este urmată de o scriere în același client
  - o scriere a unui obiect este urmată de o citire a aceluiași obiect în orice client
- Operațiile de scriere care sunt potențial cauzale trebuie văzute de toate nodurile în aceeași ordine.
- Scrierile concurente este permis să fie văzute în ordine diferită pe replici diferite.

Consistența cauzală

a) Violarea consistenței cauzale – scrierea din P1 este în relație cauzală cu scrierea din P2și de aceea, trebuie văzute în aceeași ordine de P3 și P4
b) O stare cauzală consistentă: citirea a fost eliminată și acum scrierile devin concurente. Citirile din P3 și P4 respectă regula.

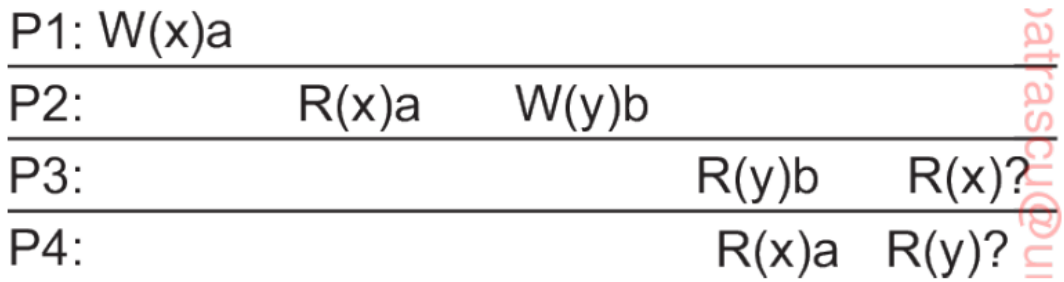


(a)



(b)

Consistența cauzală



Operația $R_3(x)$

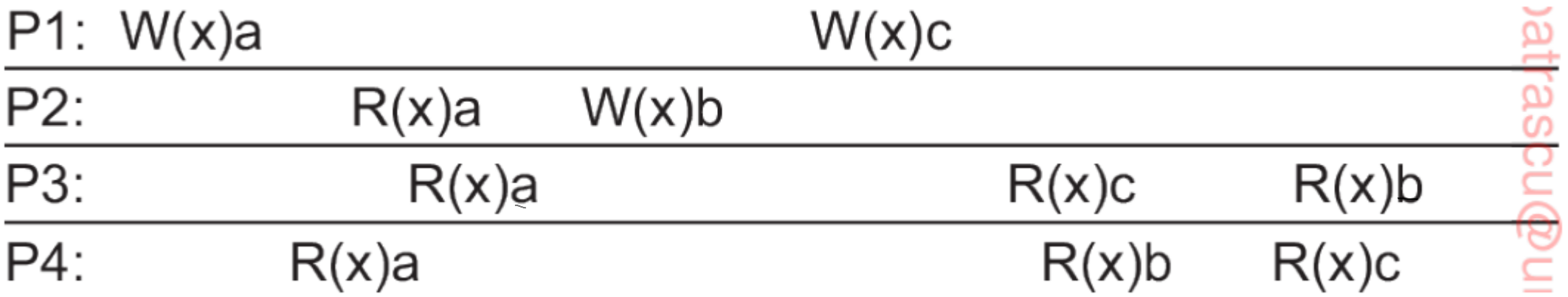
- P_3 execută $R_3(x)$ după $R_3(y)b$
- Observăm ordinea cauzală a operațiilor $W_1(x)a \rightarrow R_2(x)a \rightarrow W_2(y)b \rightarrow R_3(y)b$
- Pentru păstrarea consistenței cauzale este necesar ca $R_3(x) = R_3(x)a$

Operația $\mathrm{R}_4(x)$

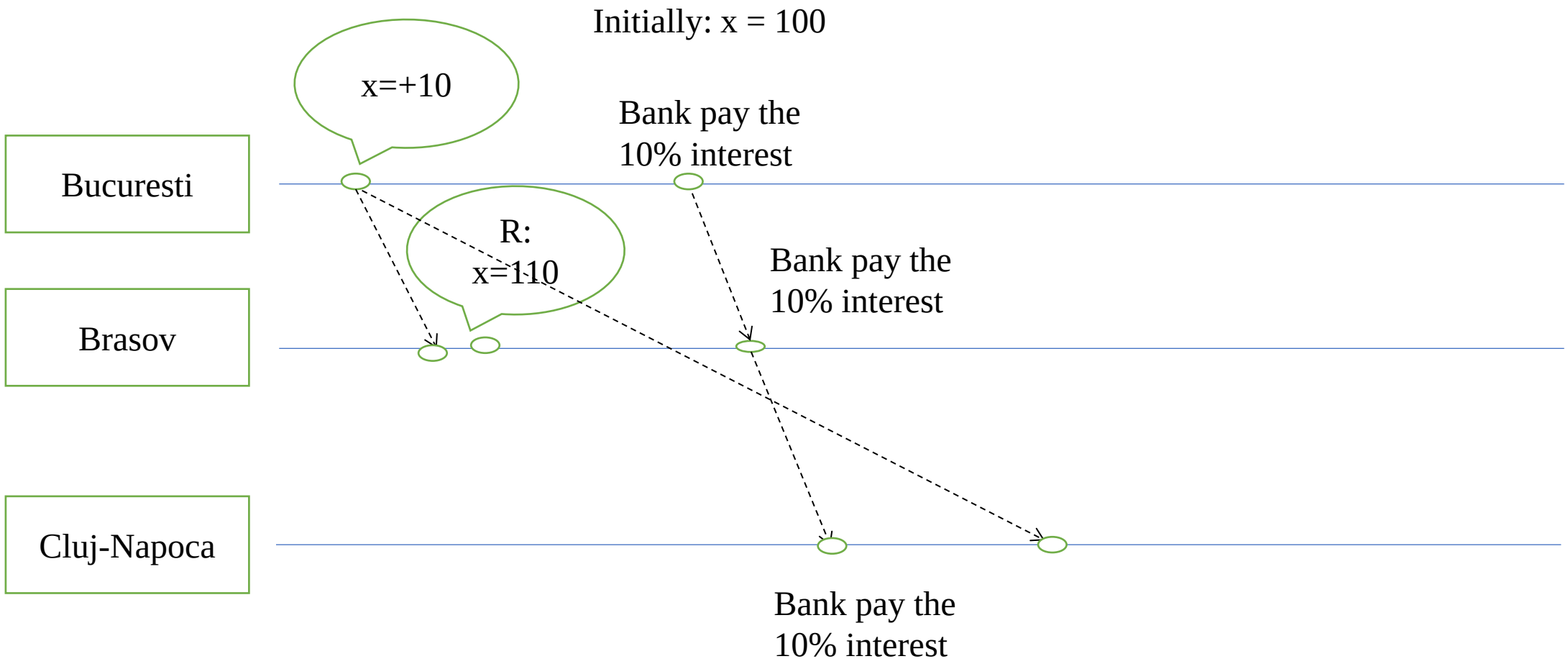
- Cu toate că avem formal relația $W_1(x)a \rightarrow W_2(y)b$, inițializările variabilelor sunt independente.
- De aceea, $\mathrm{R}_4(x)NULL$ se conformează consistenței cauzale.

Exercițiu

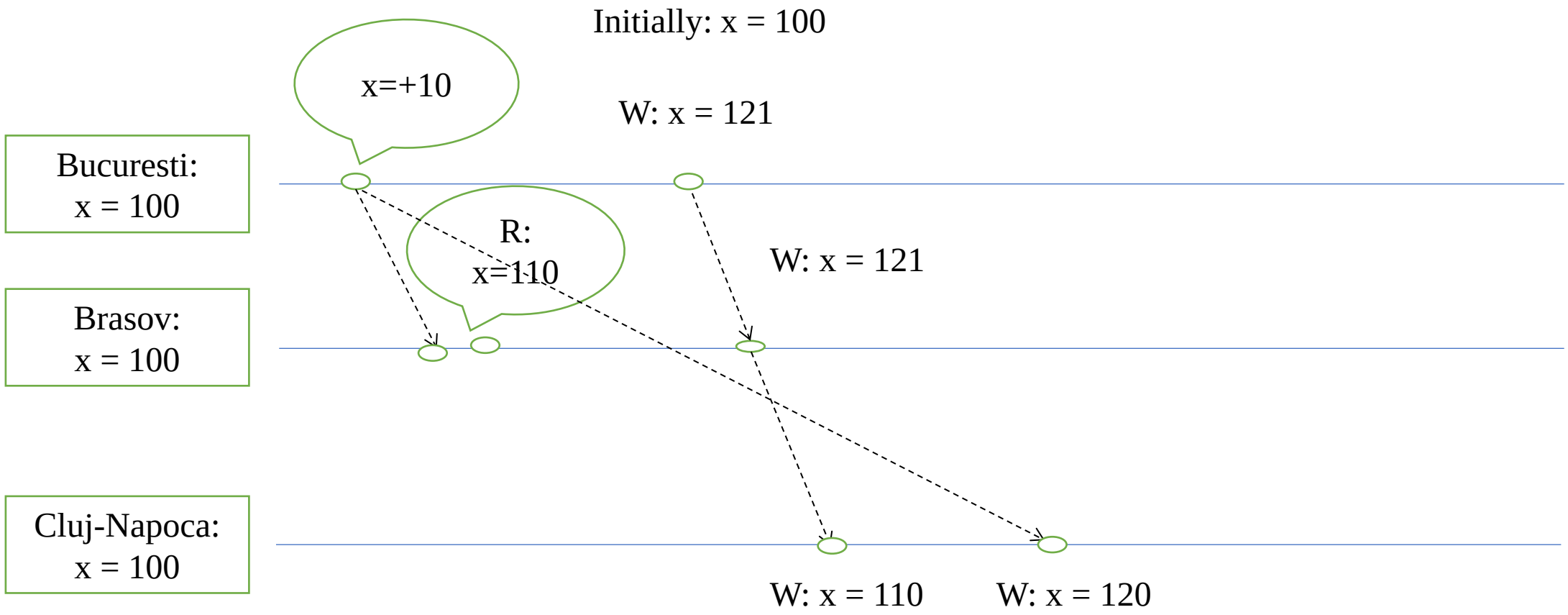
Care model de consistență se respectă în următorul scenariu?



Consistență cauzală

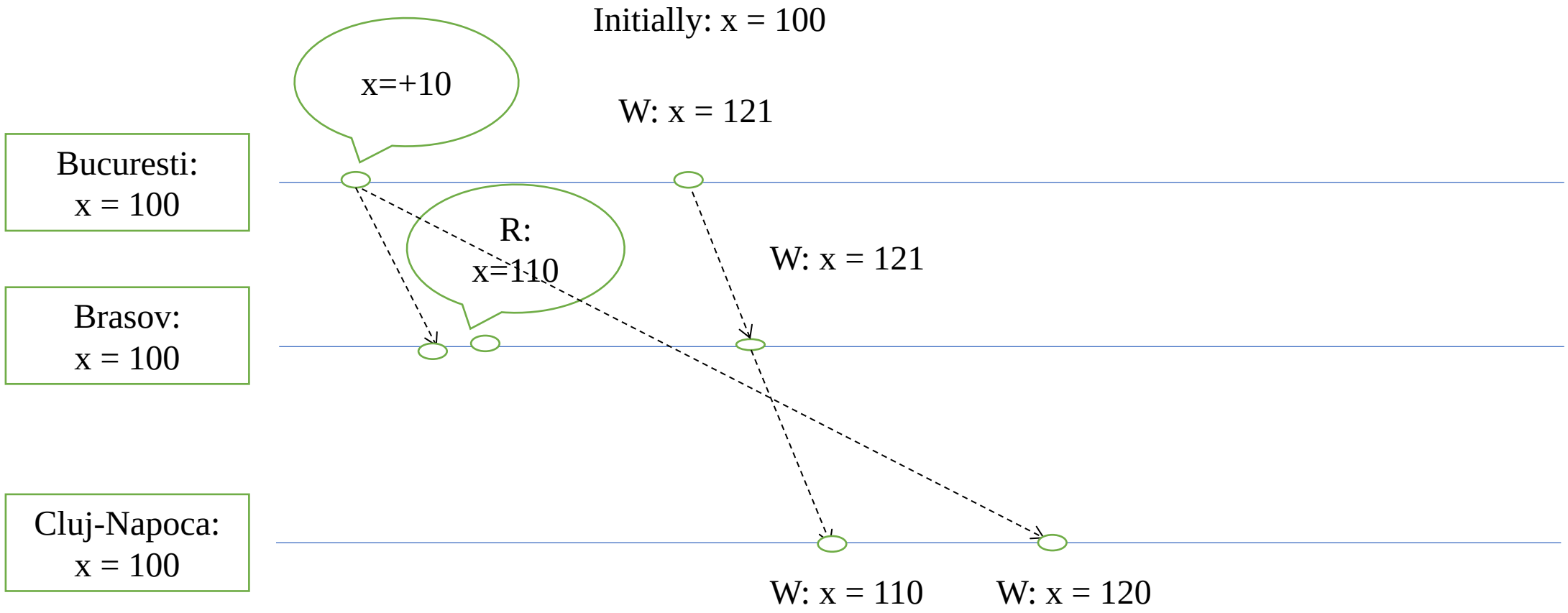


Consistență cauzală



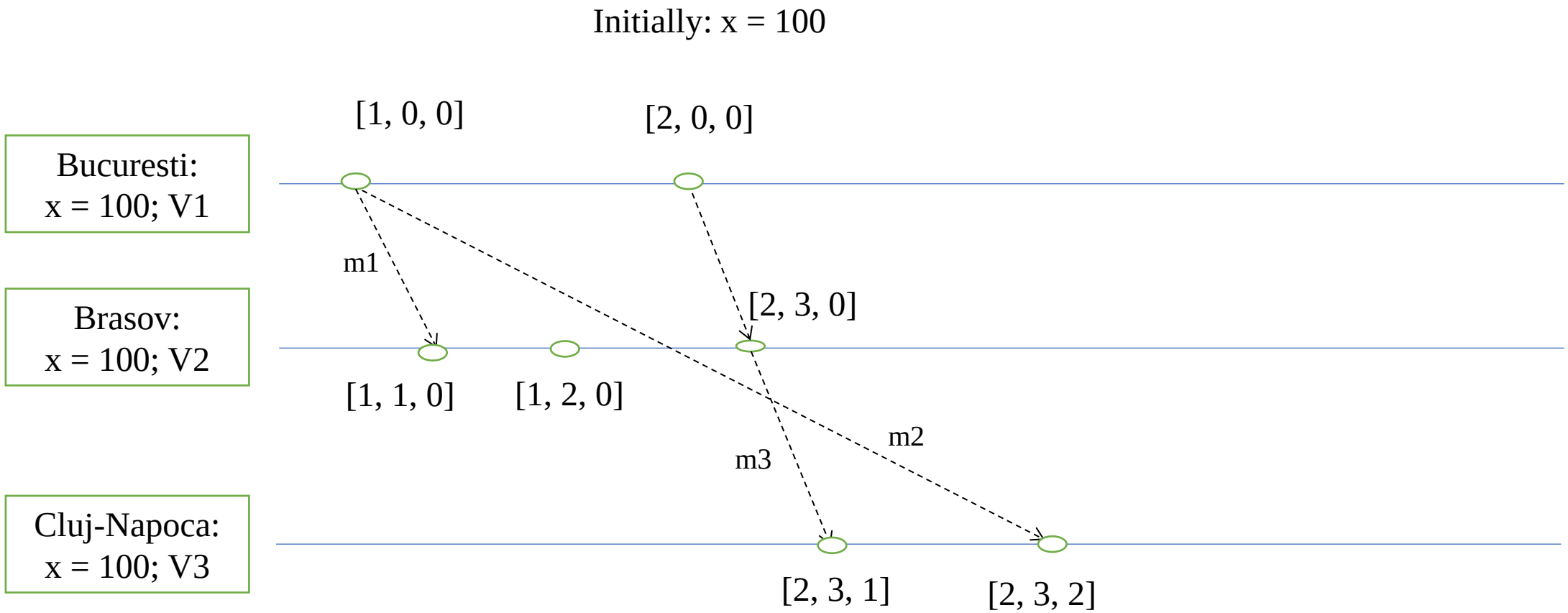
In third replica, the final sum results in 120 since it attempts to realize the write operations in reversed order; What to do?

Consistență cauzală



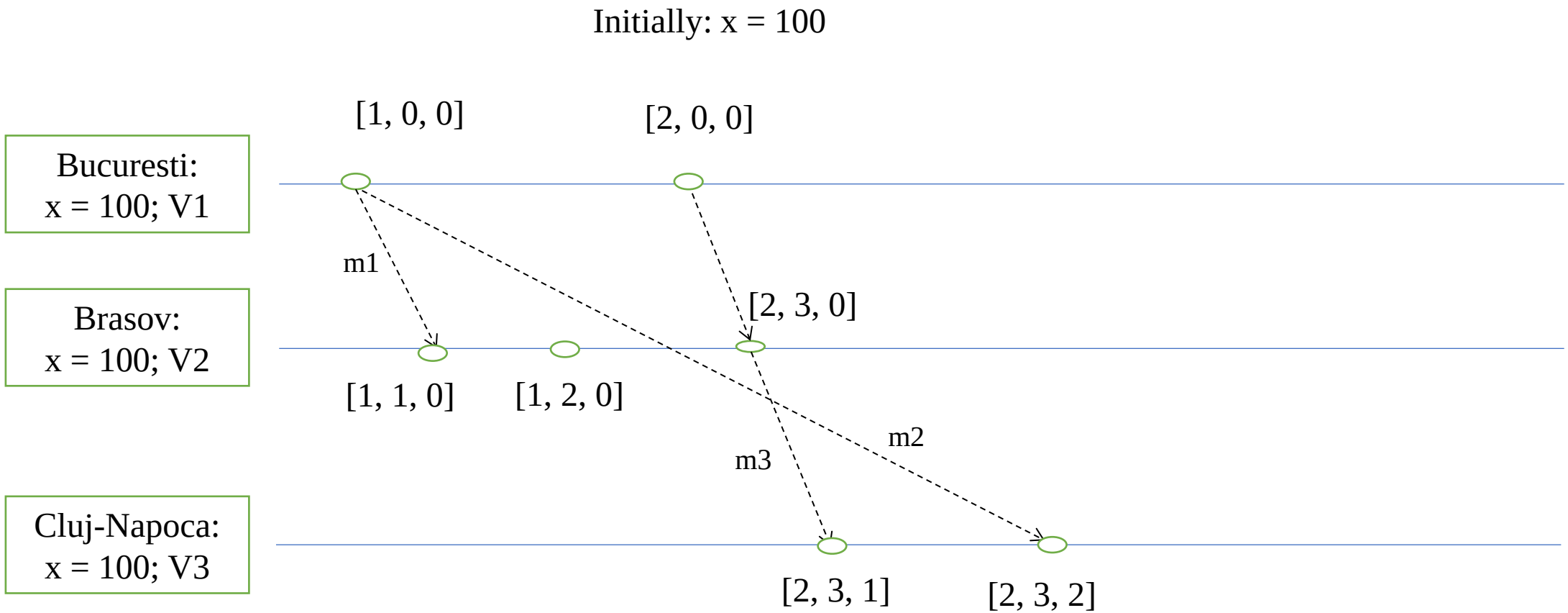
In third replica, the final sum results in 120 since it attempts to realize the write operations in reversed order; What to do? **Vector clocks**

Consistență cauzală



Timestamps: ts(m1) = [1, 0, 0]; ts(m2) = [1, 0, 0]; ts(m3) = [2, 3, 0]
Vector clocks: V1(send(m2)) < V2(send(m3))

Consistență cauzală

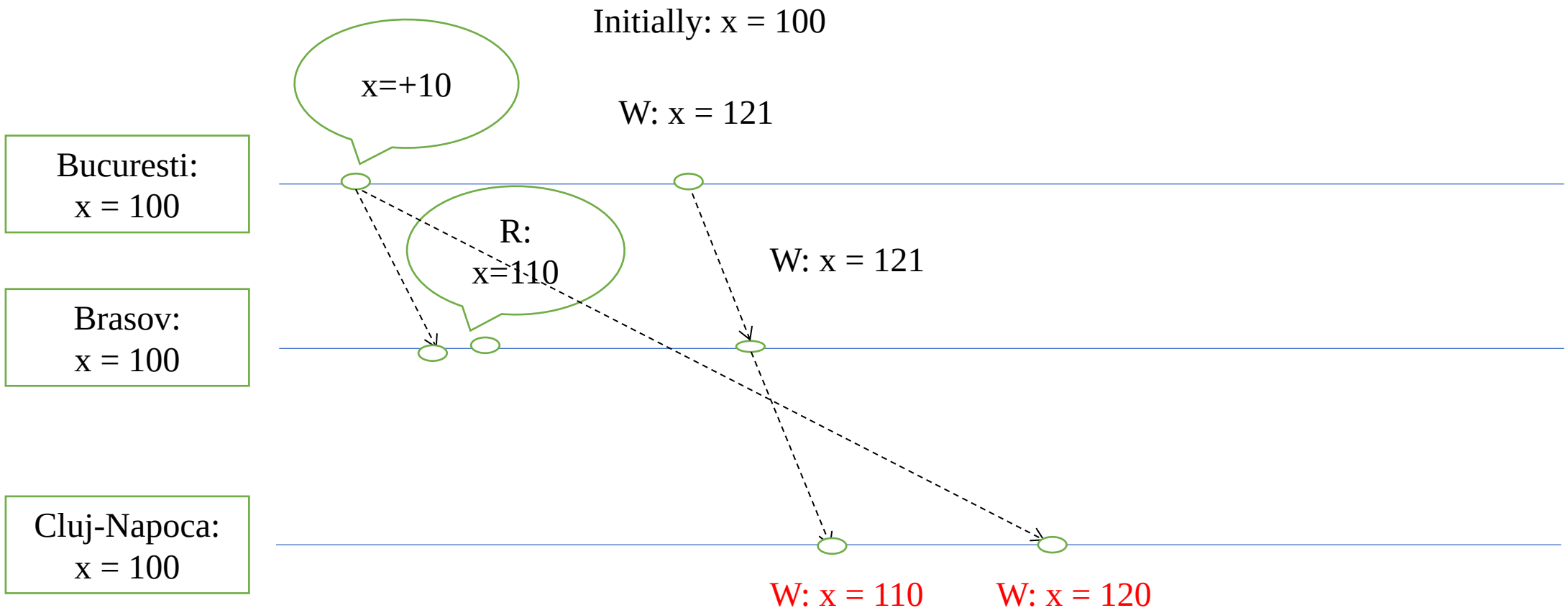


Timestamps: ts(m1) = [1, 0, 0]; ts(m2) = [1, 0, 0]; ts(m3) = [2, 3, 0]
Vector clocks: V1(send(m2)) < V2(send(m3)), therefore

m2 update operation happened before m3 update operation

m2 update causally precedes m3 update

Consistență cauzală



Third replica can easily determine that its write operations have wrong order based on vector clocks

Consistență eventuală

Sub concurență slabă, cerințele de consistență sunt slabe.

Ipoteză: Un singur nod (sau un grup redus) are dreptul să execute actualizări pe date.

- Exemplu: o pagină web este actualizată doar de către administrator (sau de către proprietar)
- Dacă nu au loc actualizări pe termen lung, atunci replicile converg la aceeași stare și devin consistente.

Cuprins

Algoritmi

Două scheme simple pentru a păstra consistența secvențială:

- **Scheme bazate pe replică primară**: fiecare element are o replică primară pe care toate operațiile de scriere sunt executate
  - Remote-write: operațiile de scriere sunt posibil executate pe o replică distantă
  - Local-write: operațiile de scriere sunt întotdeauna executate pe o replică locală.
- **Scheme bazate pe replicarea operației**: operațiile de scriere sunt executate pe mai multe replici simultan.

Schema Remote-write

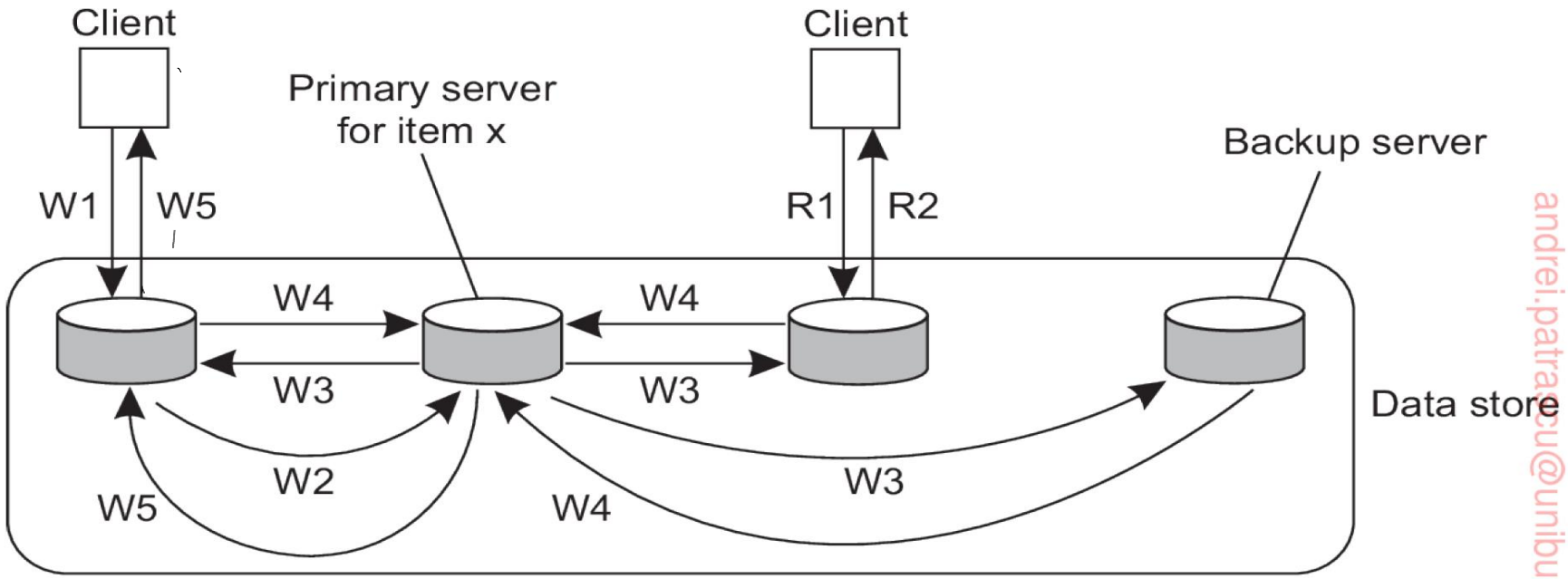
Toate operațiile de scriere sunt executate pe un singur nod-server (distant).

Acest model este asociat cu arhitecturile tradiționale client-server.

Algoritm:

1. Permite citirea locală a unui element x, trimite operația de scriere la replica primară (responsabilă de x).
2. *Blocant*: Blochează starea pe operația de scriere până toate replicile au actualizat propria copie locală
3. *Nonblocant*: Replica primară returnează și confirmă (ACK) actualizarea copiei sale locale (pentru accelerare)

Schema Remote-write



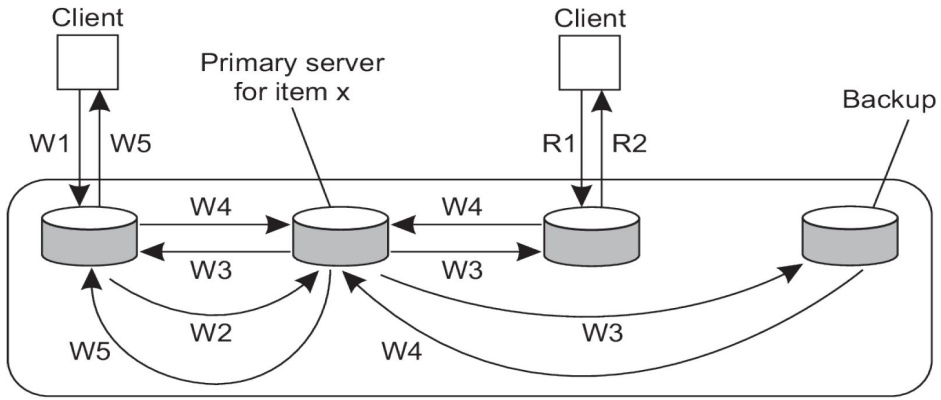
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Schema Remote-write

Specific aplicațiilor de tipul *online repository*, în care operațiile de scriere au loc online (la distanță)
e.g.

- OneDrive
- Google Drive



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

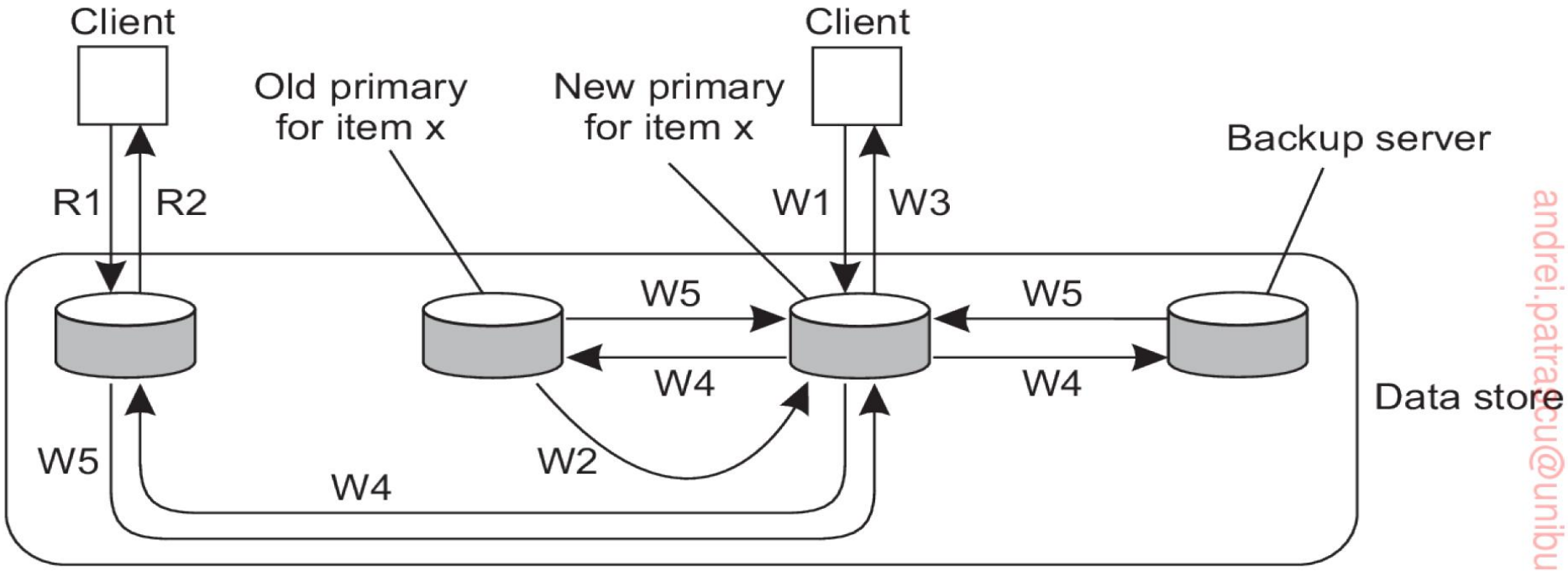
R1. Read request
R2. Response to read

Schema Local-write

O singură copie a elementului x este actualizată.

- La operația de scriere, elementul x va fi transferat la replica care realizează operațiile de scriere (primary)
  - Sunt posibile multiple scrieri succesive executate local
- Starea de “primară” a unei replici este transferabilă

Schema Local-write



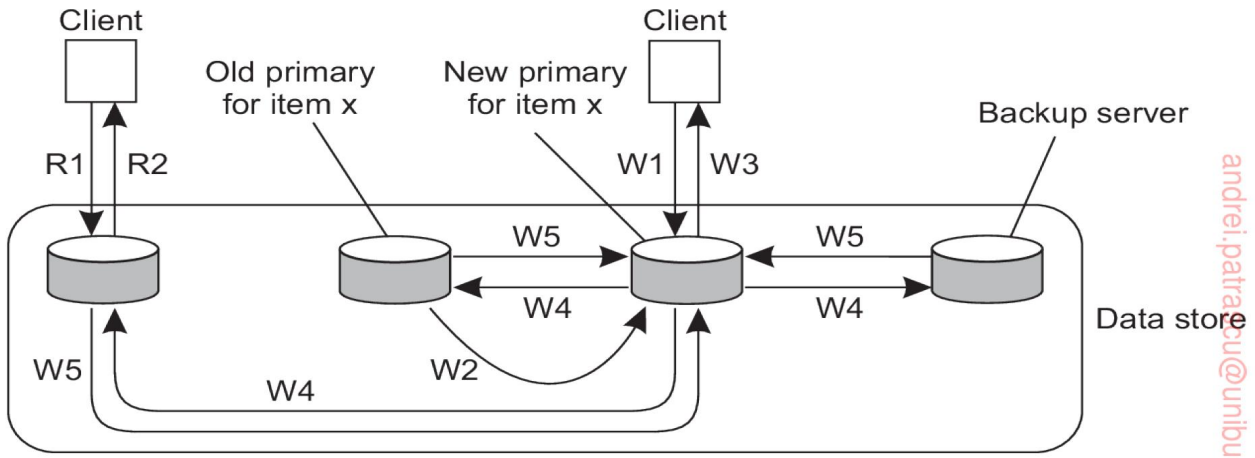
W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Schema Local-write

Specific aplicațiilor de tipul *offline repository,* în care operațiile de scriere au loc local, iar dupa conectarea la rețea se manifestă și în celelalte replici, e.g.

- Dropbox
- Git



Scheme bazate pe difuzare

În cazul în care nu avem o autoritate primară, operațiile de actualizare vor fi trimise tuturor replicilor (citiri locale).

Operațiile de scriere trebuie executate în aceeași ordine pe toate nodurile.

Algoritm de difuzare cu ordonare totală:

1. P_i face Bcast(w)
2. Dacă P_j recv(w), mesajul este pus în coadă, ordonat după marcajul de timp; reply(ACK) către sursă
3. Se realizează operația dacă este prima din coadă și sunt prezente semnale ACK de la toate celelalte noduri.
4. După execuția operației, se elimină din coadă (împreună cu ACK aferente)

Scheme bazate pe difuzare

- Dacă se menține un ceas Lamport local atunci $C(m) < C(ACK)$
- Asimptotic, cozile din fiecare nod vor deveni identice
- Pentru execuția unei scrieri sunt necesare minim $O(n)$ mesaje

Cuprins

Excludere Mutuală

Procesele unui SD adesea necesită acces la aceeași resursă, i.e. acces mutual exclusiv la resurse partajate.

Algoritm de excludere mutuală *= metoda pentru evitarea folosirii simultane a unei resurse comune (e.g. o variabilă globală).*

- *Centralizată (sisteme de operare)*
- *Distribuită*

Excludere mutuală centralizată

Ipoteze:

- Pp. un număr de procese (unul este coordonator)
- Fiecare proces necesită confirmarea cu coordonatorul înaintea intrării în “secțiunea critică”

Proces (non-coordonator)

Pentru a obține acces: send *request*, await *reply*

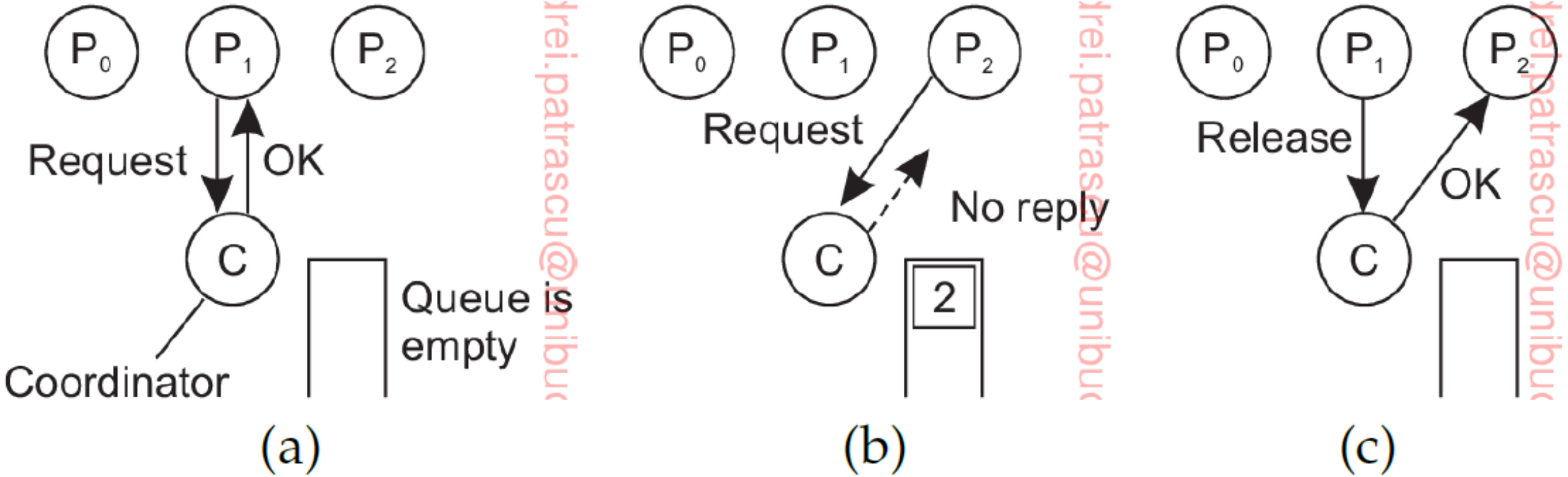
Pentru ieșire (release): send *release_message*

Coordonator:

Primește request*:* dacă resursa este valabilă și coada goală atunci send OK; altfel request pentru coadă;

Primește release*:* elimină următorul request din coadă și *send OK*

Excludere mutuală centralizată



Excludere mutuală centralizată

Avantaje:

- *Echitabilitate:* semnalele request sunt respectate în ordinea primirii
- *Simplitate:* trei mesaje pentru folosirea unei resurse
- Nu apare „înfometarea” (starvation) proceselor: nu există proces care solicită accesul și nu-l va primi până la incheierea algoritmului.

Dezavantaje:

- Coordonatorul este punct vulnerabil de defect. *Cum detectăm un coordonator defect?*
- Când $n \to \infty$, performanța scade

Excludere mutuală distribuită

Idee: Putem folosi ceasurile logice Lamport pentru ordonarea solicitărilor?

Premisă: Fiecare proces P_i păstrează un ceas logic L_i.

Algoritmul Ricart-Agrawala:

1. Când P_i intră în secțiunea critică:
    1. Incrementează: $L_i = L_i + 1$.
    2. Difuzează (L_i, i) către toate $P_j, j \neq i$
    3. Așteaptă reply de la celelalte procese.
    4. Intră în secțiunea critică.

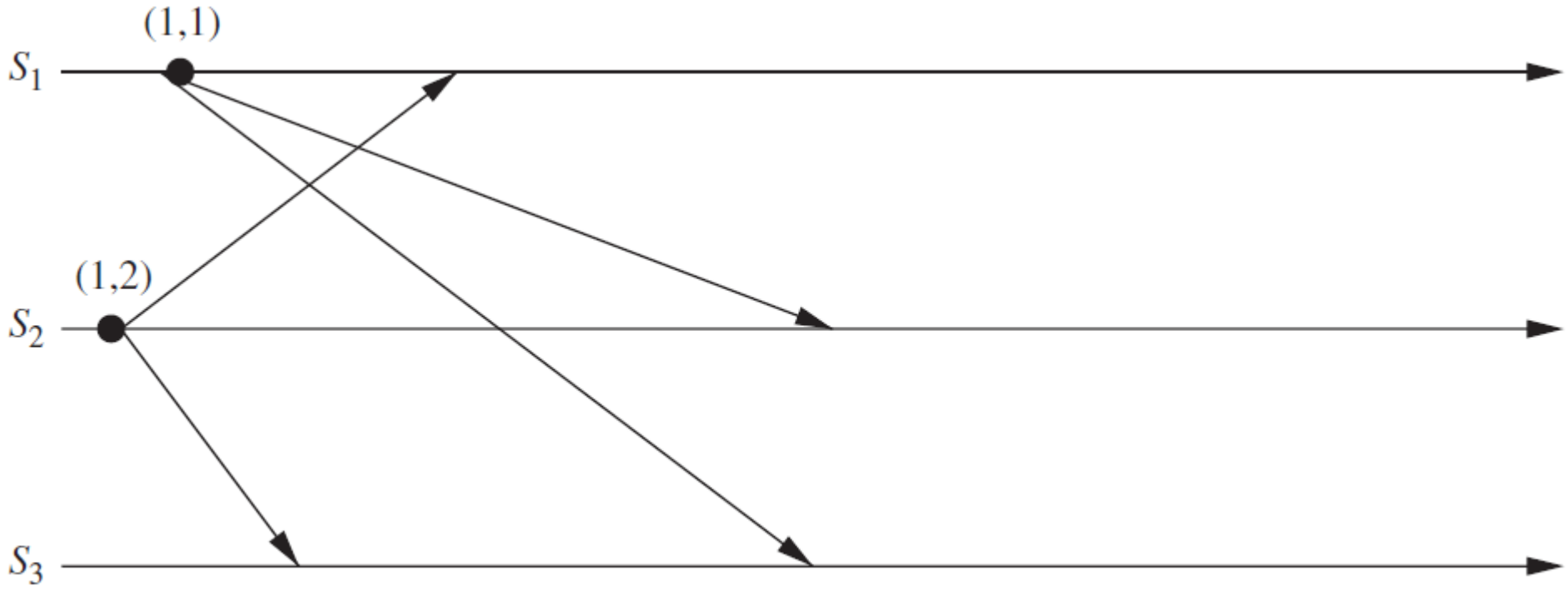
Excludere mutuală distribuită

Premisă: Fiecare proces P_i păstrează un ceas logic L_i.

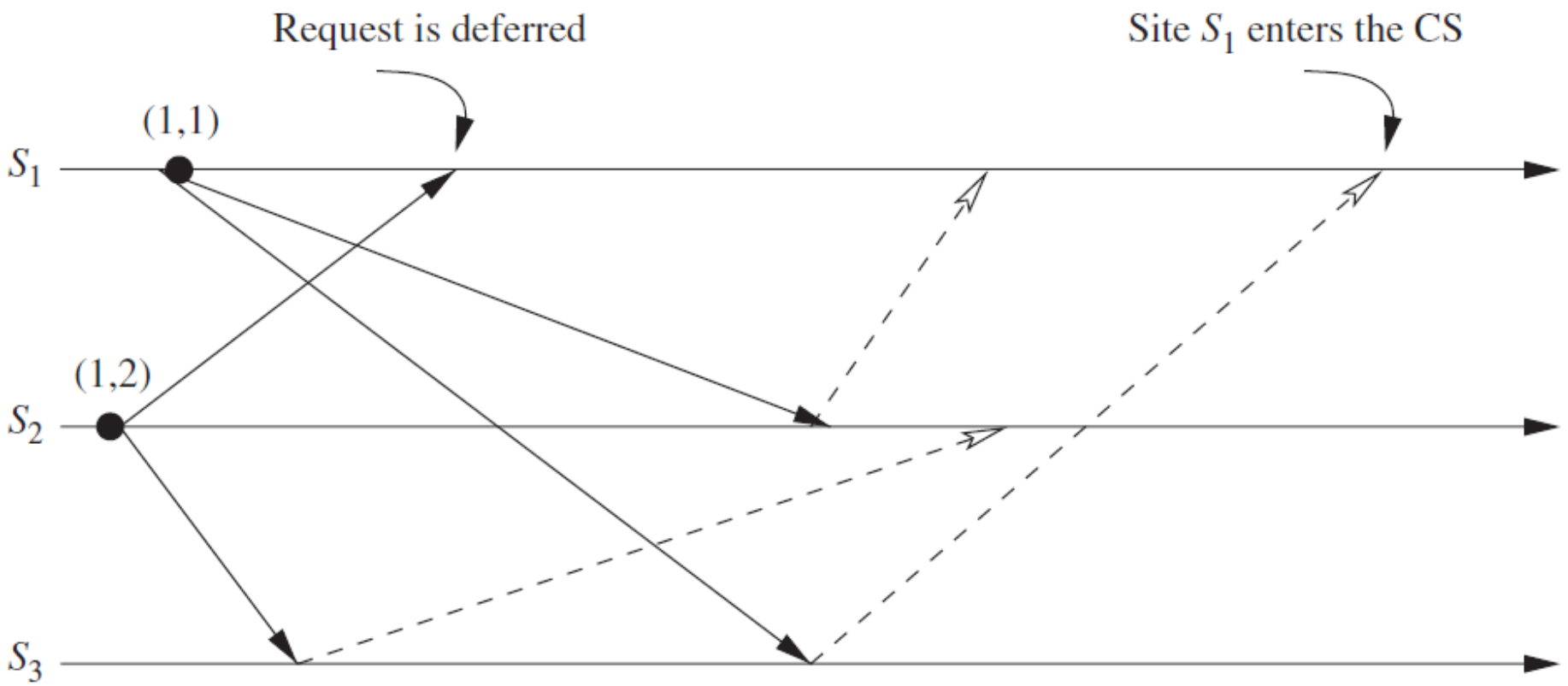
2. Când P_j primește un mesaj de la P_i:
    1. Dacă se află în afara secțiunii critice: *send OK*.
    2. Dacă se află în secțiunea critică: nu răspunde, adaugă *request* în coadă.
    3. Dacă intenționează să intre în secțiunea critică:
        1. dacă $(L_i, i) < (L_j, j)$: *send OK*
        2. altfel: adaugă *request* în coadă.

3. Când P_i finalizează ocuparea secțiunii critice, difuzează OK către procesele din coada sa.

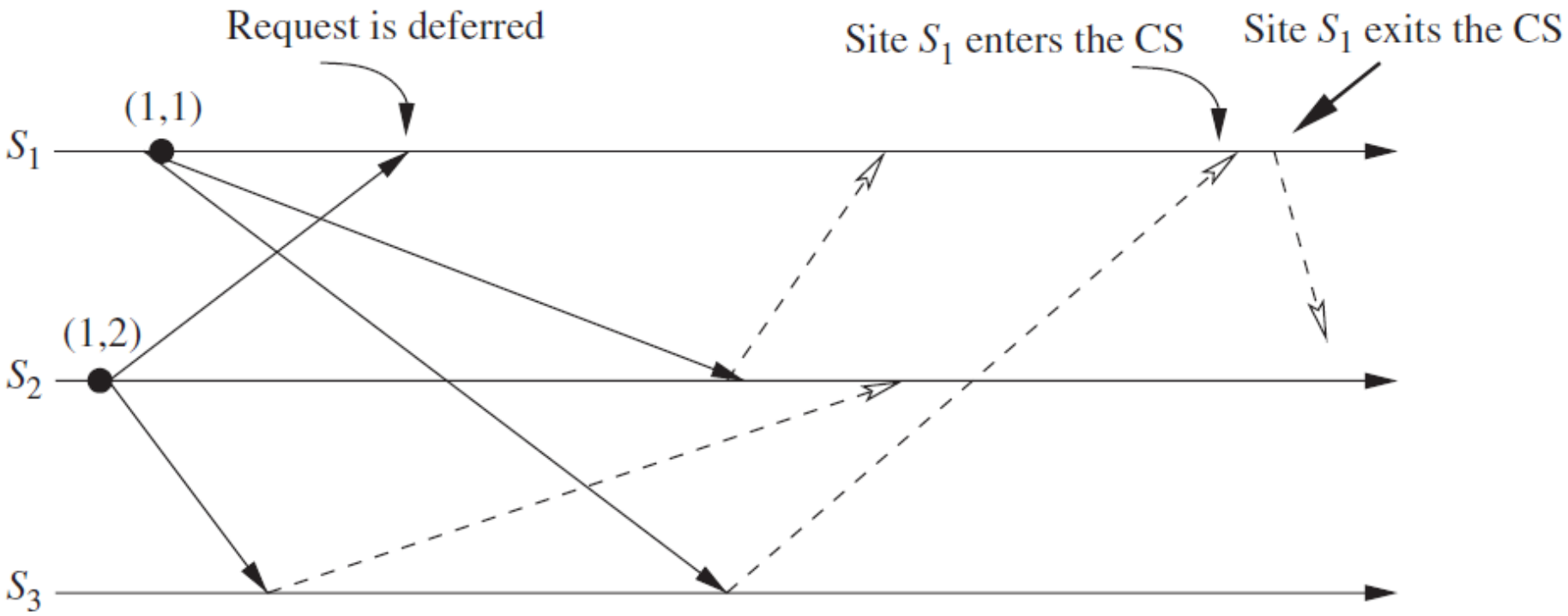
Excludere mutuală distribuită



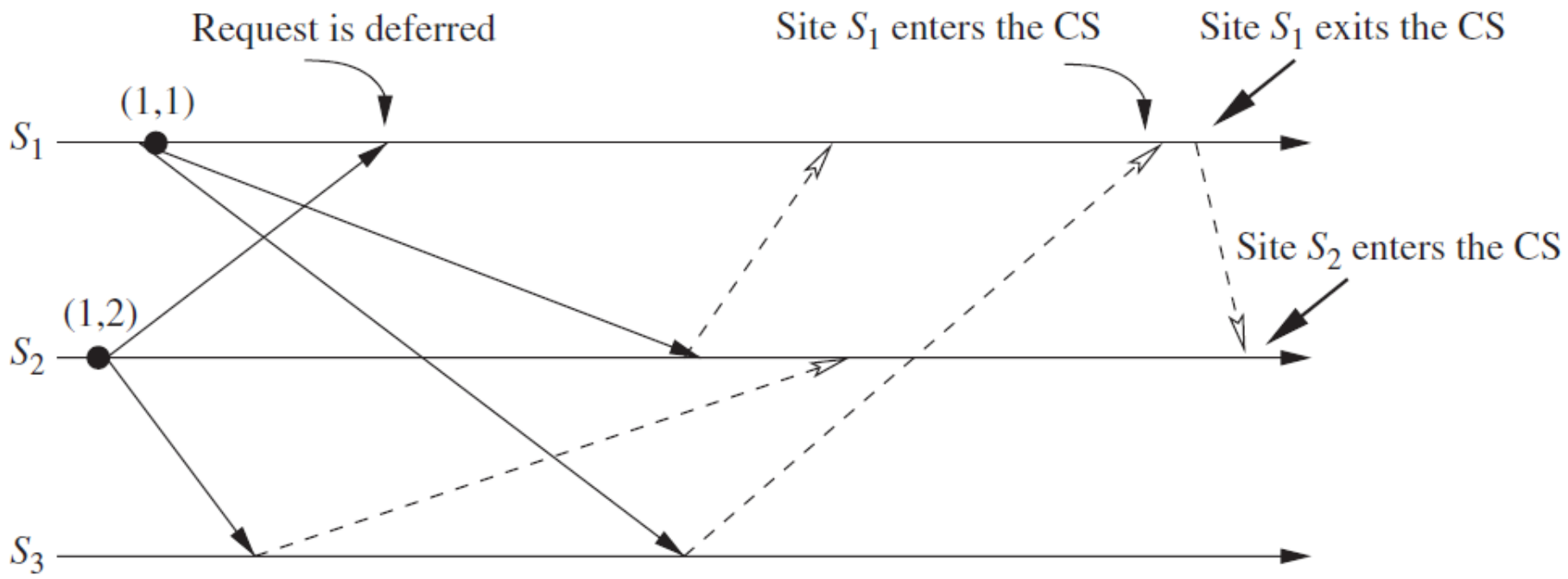
Excludere mutuală distribuită



Excludere mutuală distribuită



Excludere mutuală distribuită



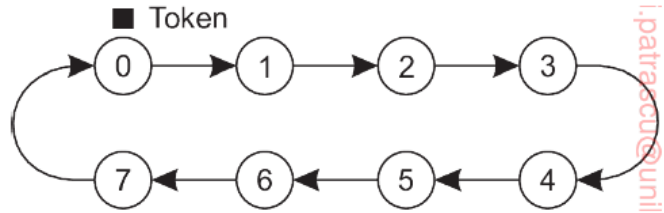
Excludere mutuală distribuită

Analiză:

- Toate procesele sunt implicate în toate deciziile
- Necesită $2(N-1)$ mesaje per intrare în secțiunea critică
- Dacă apar defecte (crash), schema trebuie completată cu semnale care să faciliteze distincția între starea de defect și dezacordul legate de intrarea în s.c.
- Îmbunătățire: P_i întră în s.c. când permisiunea de la majoritatea nodurilor.

Excludere mutuală bazată pe jeton

- Un jeton unic este partajat între nodurile sistemului.
- Unui nod i se permite intrarea în SC dacă are jetonul.
- Un nod poate intra în SC de mai multe ori până pasează jetonul.
- În acest caz, obținerea EM este trivială!



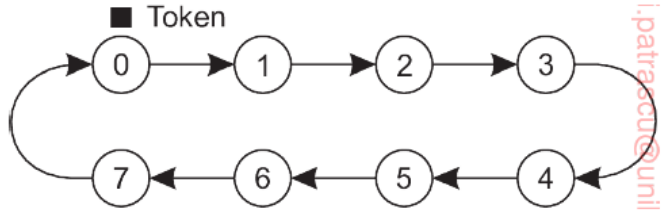
Algoritm Token Ring

- P_i comunică cu vecinii

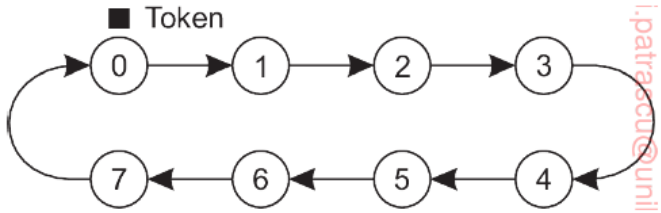
1. P_0 primește inițial jetonul pentru a accesa resursa R
2. Pasează jetonul mai departe: P_i trimite jetonul către $P_{\{i \bmod N\}}$
3. Procesele așteaptă jetonul pentru a intra în SC

Când P_i primește jetonul:

- Dacă așteaptă intrarea în SC, oprește jetonul până la ieșire
- Altfel, pasează jetonul mai departe



Algoritm Token Ring



- Ordonarea bine definită a intrării în SC
- Dacă jetonul se pierde (prin defect), poate fi regenerat (problemă netrivială)
- Nodurile defecte întrerup inelul.
- Dezavantaj: Apar diferențe de răspuns în situațiile (T durată livrare token, E timp execuție SC, n noduri):
  - Încărcarea este **slabă**
  - Încărcarea este **puternică**