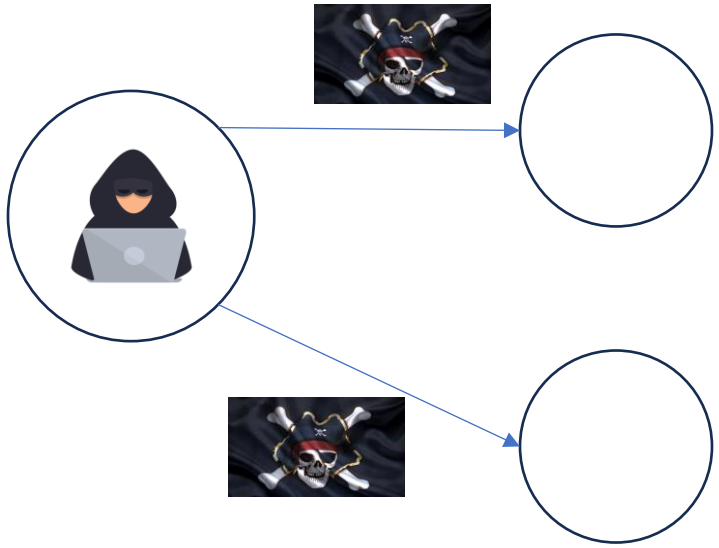
Sisteme și algoritmi distribuiți
Curs 7

Defecte bizantine (reluare)

Defect Bizantin

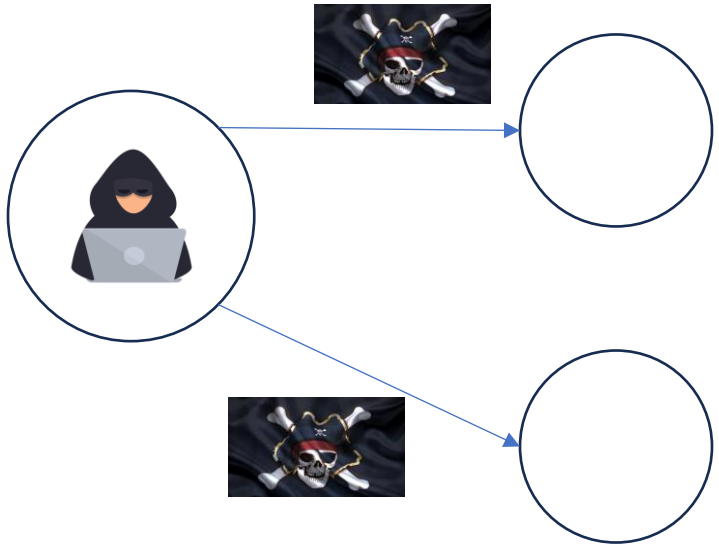
Sub defect bizantin nodurile se comportă malițios, perturbând activitatea întregului sistem (e.g. comportament arbitrar):

- Livrează mesaje atipice execuției algoritmului local
- Actualizează starea după reguli atipice execuției algoritmului local



Defect Bizantin

Teorema [Lynch]. Într-un sistem distribuit sunt necesare $\boldsymbol{n > 3s}$ procese pentru toleranța a s defecte bizantine.



Consens distribuit (cu procese defecte)

Starea (valoarea) uniformă a nodurilor unui sistem distribuit.

Condiția de acord: Nu există două procese corecte care decid valori diferite.

Condiția de validitate: Dacă valoarea inițială a proceselor este v, atunci consensul se atinge cu valoarea uniformă v.

Condiția de terminare (algoritm): Într-un algoritm de consens, orice nod *corect* din sistem va decide eventual la un moment de timp.

În general, decizia se reduce la evaluarea funcției de consens f(·) în $x(0)$.

Consens distribuit (cu procese defecte)

Ipoteze:

- Sistem cu n procese si maxim s defecte (bizantine)
- Pentru început, considerăm graf *complet* al sistemului.
- Presupunem că numărul de noduri din sistem satisface: $n > 4s$.
- Fiecare nod își cunoaște indexul în sistem.

Algoritmul ByzFlood

Algoritm **ByzFlood**(Maj()):

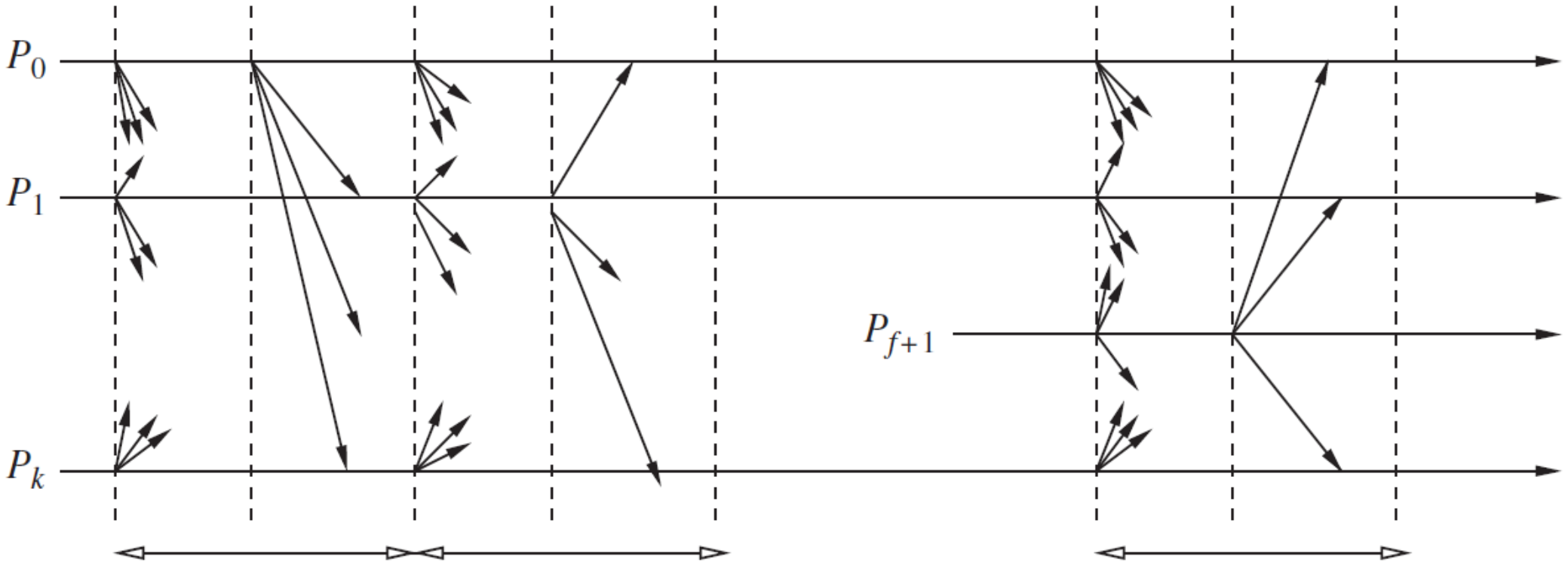
M_i: - int $x(0)$ (starea inițială, inițial egal cu v_i)
- int *s*, integer (număr maxim de defecte)
- int *t*, integer, inițial 0

% Fiecare iterație are 2 runde (t contorizează iterații)

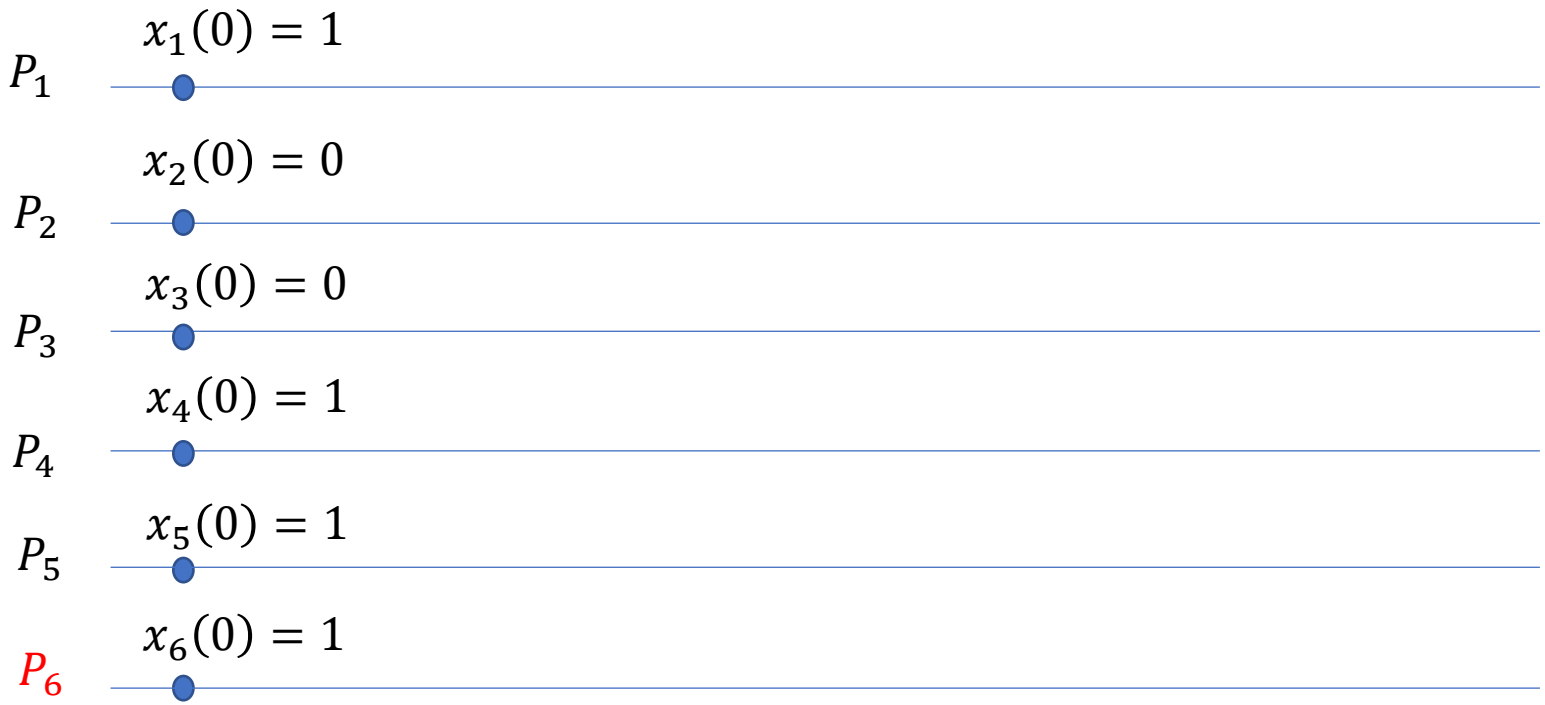
Funcție transformare nod i ():
% Runda 1
1. **Bcast**($x(0)$) % difuzează $x(0)$
2. Fie U mulțimea mesajelor $v_j = x_j(t)$ primite restul nodurilor
3. $Majority(t) = Maj(U)$
4. $mult(t) = numărul\ de\ apariții\ al\ valorii\ Majority(t)$

% Runda 2
1. **If** (i==t): % nodul leader/king
    1. **Bcast**($Majority(t)$)
2. **Else**: recv(Tie, P_t)
3. **If** (mult(t) > n/2 + s):
    1. $x(t) := Majority(t)$)
4. **Else**: $x(t) :=$ Tie
5. **If** (t>s+1):
    1. **Return** $x(t)$
6. $t := t+ 1$

Algoritmul ByzFlood

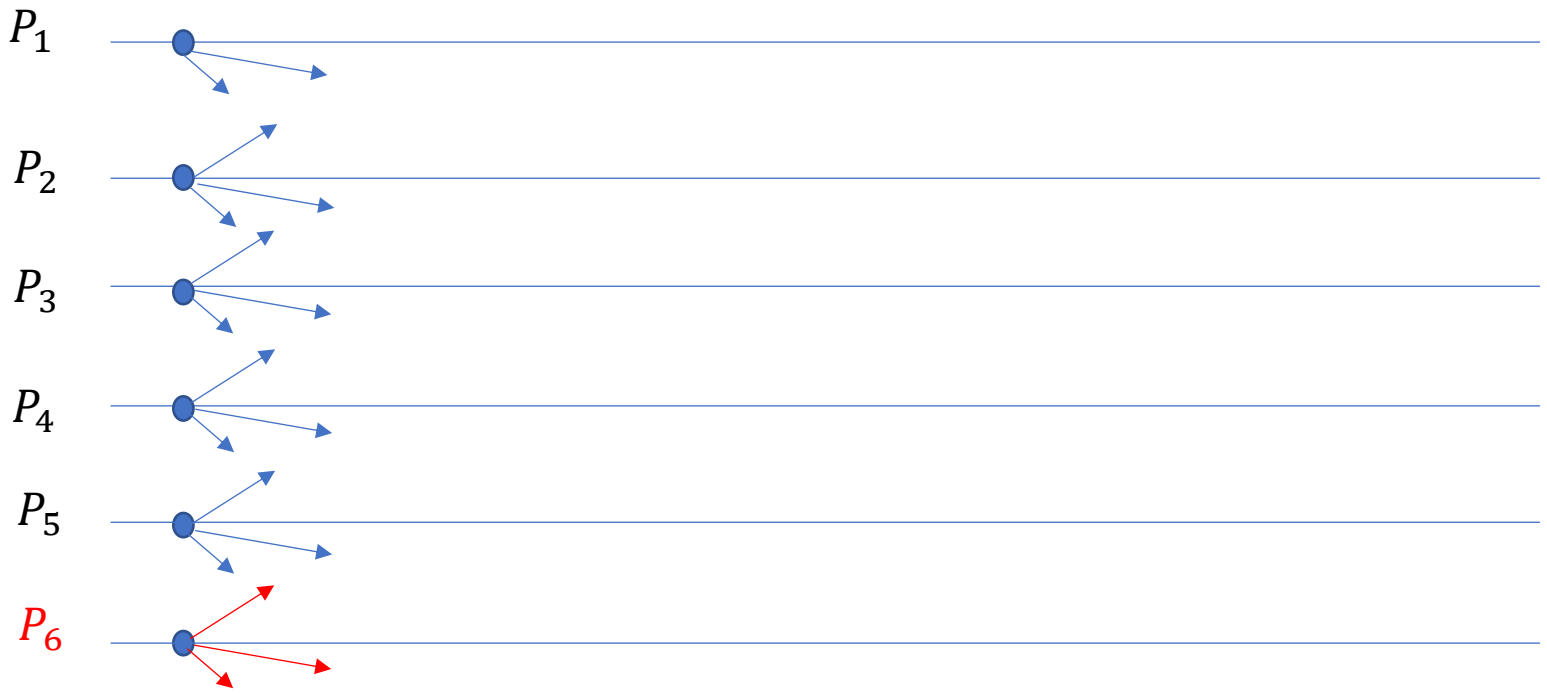


Exemplu ByzFlood(s = 1)



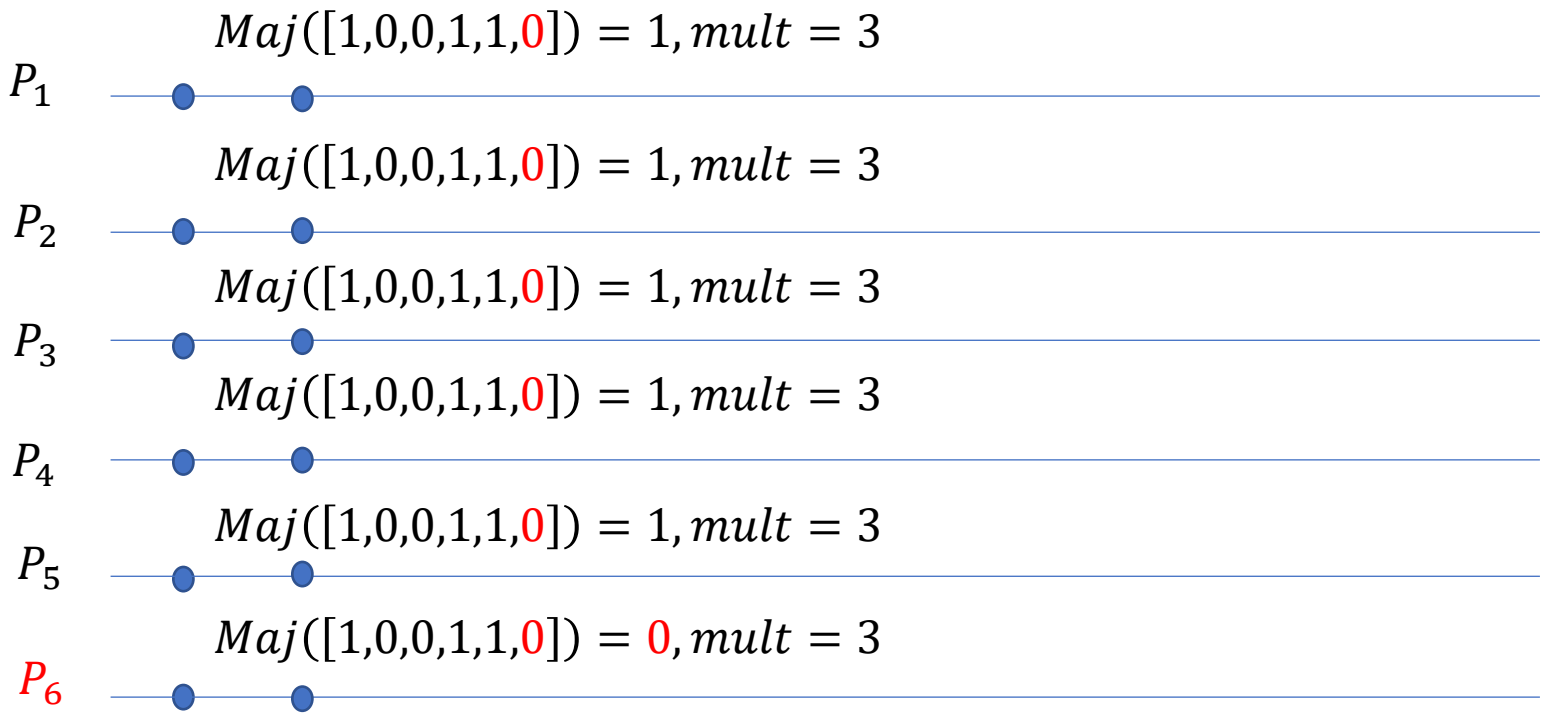
$t = 0$ $(n > 3s,\ P_6\ defect\ arbitrar)$

Exemplu ByzFlood



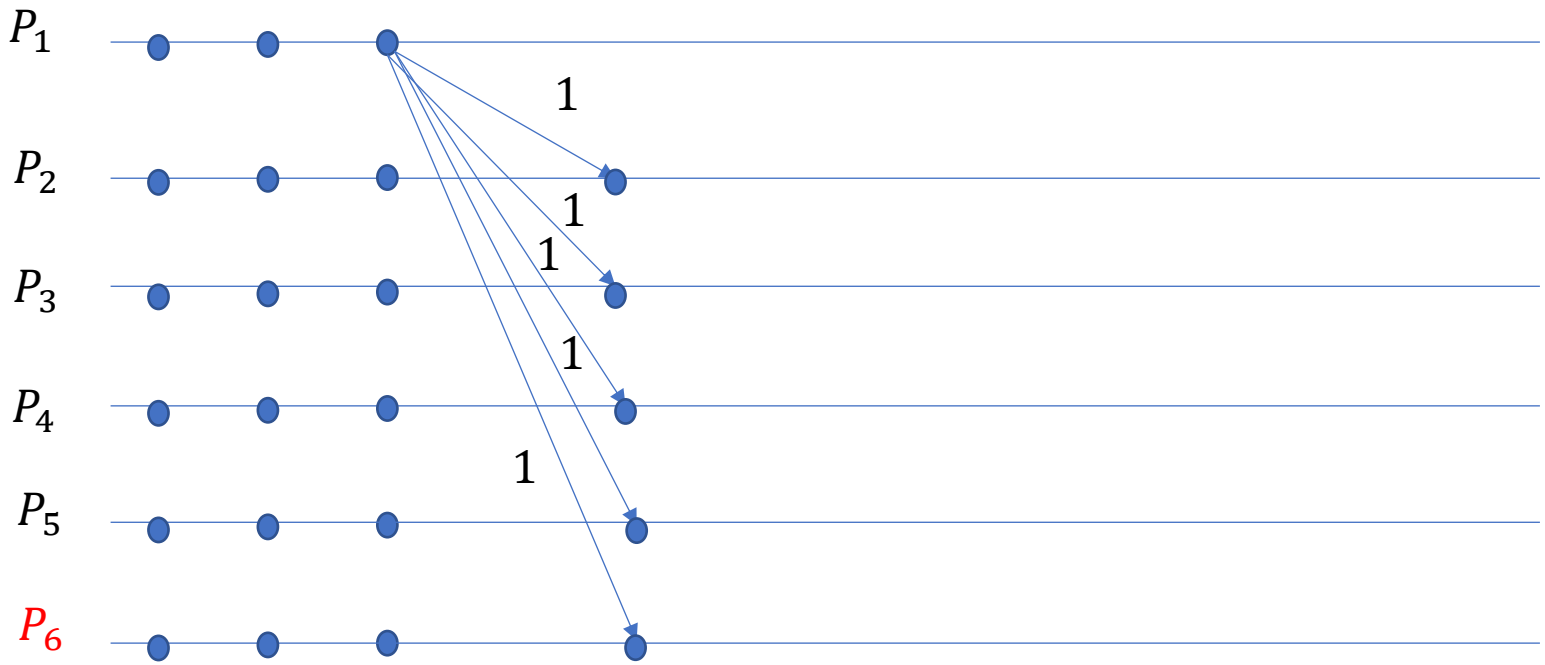
$t = 1$: $Bcast(x_i(0))$

Exemplu ByzFlood



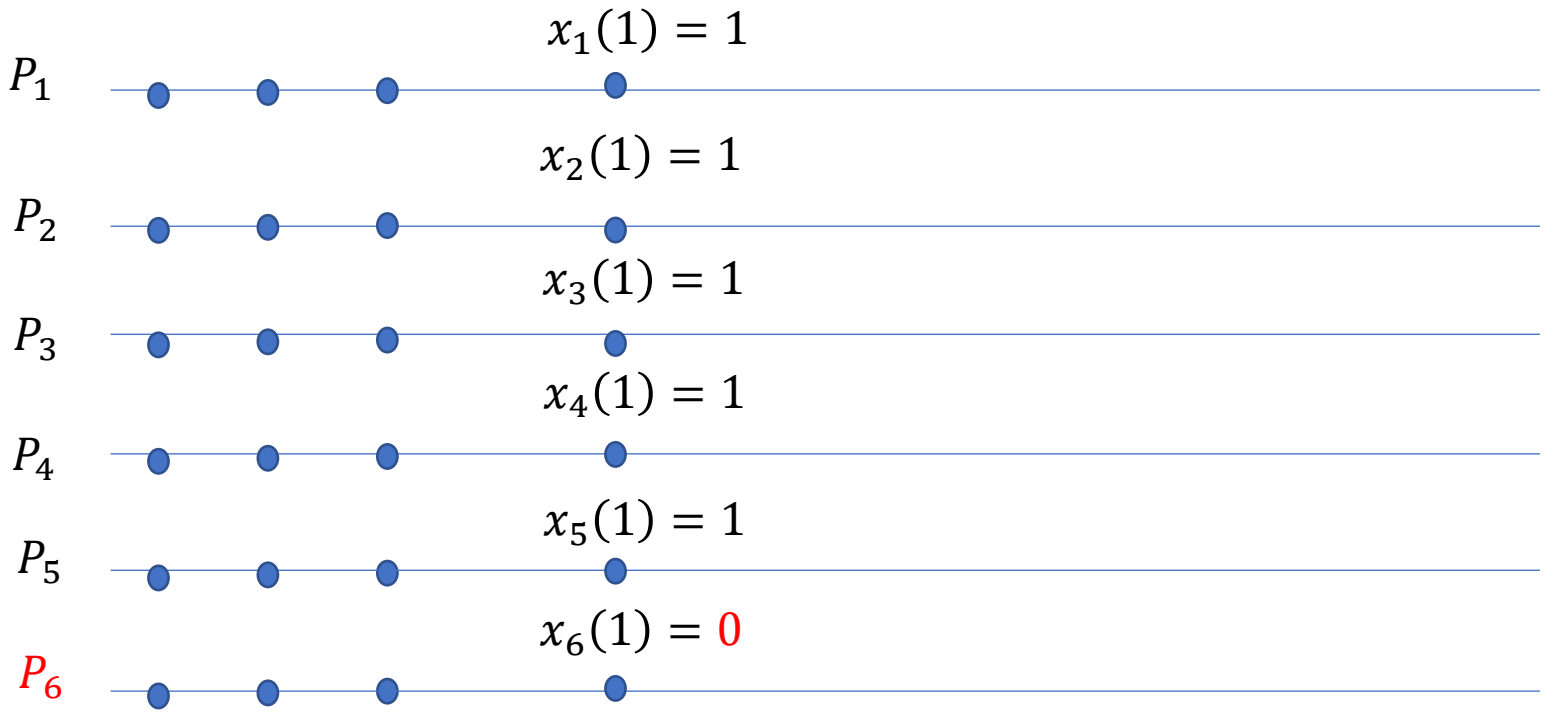
$t = 1 : Calcul\ Maj(U), mult$

Exemplu ByzFlood



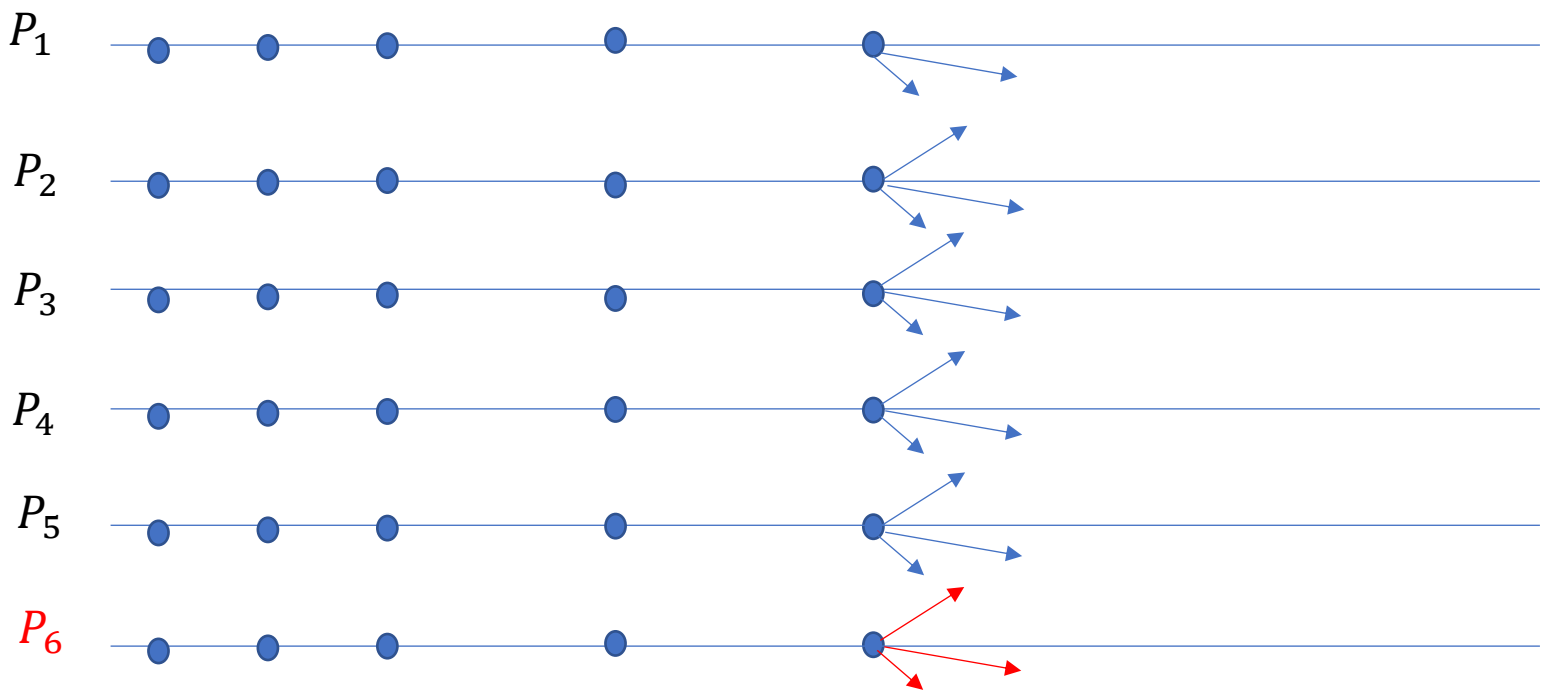
$t = 1: King \rightarrow Bcast(majority)$

Exemplu ByzFlood



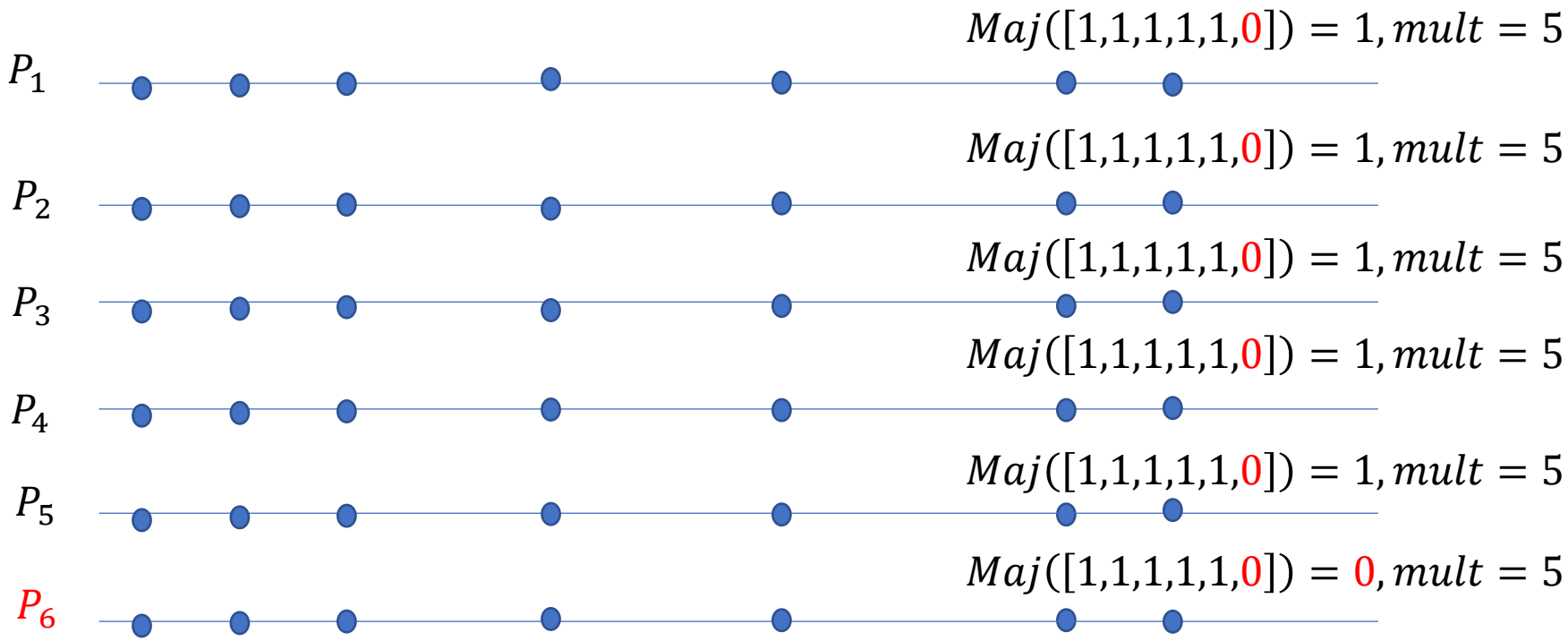
$$t = 1:\ mult < \frac{n}{2} + s = 4 \Rightarrow majority = king - tie$$

Exemplu ByzFlood



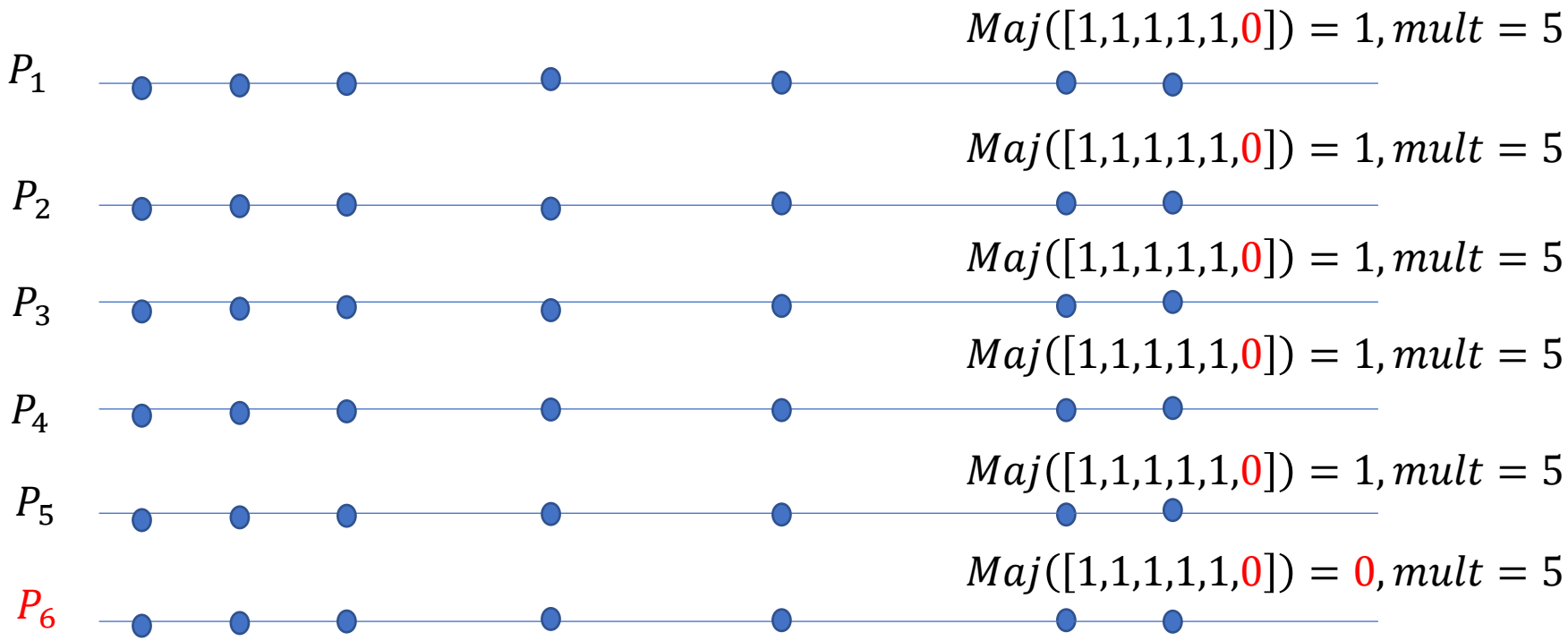
$t = 2$: $Bcast(x_i(1))$

Exemplu ByzFlood



$$t = 2 : Calcul\ Maj(U), mult = 5 > \frac{n}{2} + s = 4 \Rightarrow majority = 1$$

Exemplu ByzFlood



Consensul se obține la $t = 2$; persistă și în restul iterațiilor $s + 1 \geq t > 2$.

Algoritmul ByzFlood

Algoritm **ByzFlood**(Maj()):

M_i: - int $x(0)$ (starea inițială, inițial egal cu v_i)
- int s, integer (număr maxim de defecte)
- int *t*, integer, inițial 0

Funcție transformare nod i ():
% Runda 1
1. **Bcast**($x(0)$) % difuzează $x(0)$
2. Fie U mulțimea mesajelor $v_j = x_j(t)$ primite restul nodurilor
3. $Majority(t) = Maj(U)$
4. $mult(t) = numărul\ de\ apariții\ al\ valorii\ Majority(t)$

% Runda 2
1. **If** (i==t): % nodul leader/king
    1. **Bcast**($Majority(t)$)
2. **Else**: recv(Tie, P_t)
3. **If** (mult(t) > n/2 + s):
    1. $x(t) := Majority(t)$)
4. **Else**: $x(t) :=$ Tie
5. **If** (t>s+1):
    1. **Return** $x(t)$
6. $t := t + 1$

Analiza convergentei:
1. Între cele s+1 iterații există cel puțin una (să zicem k) în care nodul king este nod corect.
2. La iterația k, două noduri P_i și P_j se pot afla în situațiile:

- P_i și P_j actualizează x_i și x_j pe baza majorității (dacă valoarea majorității este b, atunci mult> n/2 + s; de aceea majoritatea proceselor adoptă valoare b)
- P_i și P_j actualizează x_i și x_j pe baza Tie
- P_i act. pe baza majorității și P_j actualizează pe baza Tie. P_i are mult> n/2 + s. De asemenea, și P_k are primit cel puțin n/2 voturi pentru aceeași valoare.

3. Dacă s-a atins consensul la iterația k, atunci va persista si in iteratiile subsecvente.

În cele 3 situații P_i și P_j ajung la consens.

Concluzii

- În grafuri conexe ne-complete pierdem funcția de **Bcast;** de aceea sunt necesare rutine robuste de difuzare (bazate pe istoric)
- Fără indexul de proces pierdem runda de difuzare executată de procesul king (emulare index prin mesaje speciale)
- Renunțând la ipoteze, probleme devine tot mai grea
- Alți algoritmi: Paxos (asemănător cu ByzFlood, bazat pe quorum-uri), Raft, Chandra-Toueg, Gossip, etc.
- În practică se urmărește implementarea algoritmilor specifici sistemelor *asincrone*.

Înapoi la sincronizare: Ceasuri logice și cauzalitate

Executie - traiectorie

SD supuse la defecte: posibile pierderi pe comunicația de mesaje (*packets loss*) sau defecte pe noduri (*crash-faults*). Procesele pornesc din starea inițială $x(0)$. O traiectorie/execuție este un șir (in)finit

$$x(0), e(1), x(1), e(2), x(2), \ldots \ldots$$

Ordonarea evenimentelor

Ordinea evenimentelor din traiectoria unui sistem distribuit redă influența unui proces (nod) asupra altor procese.

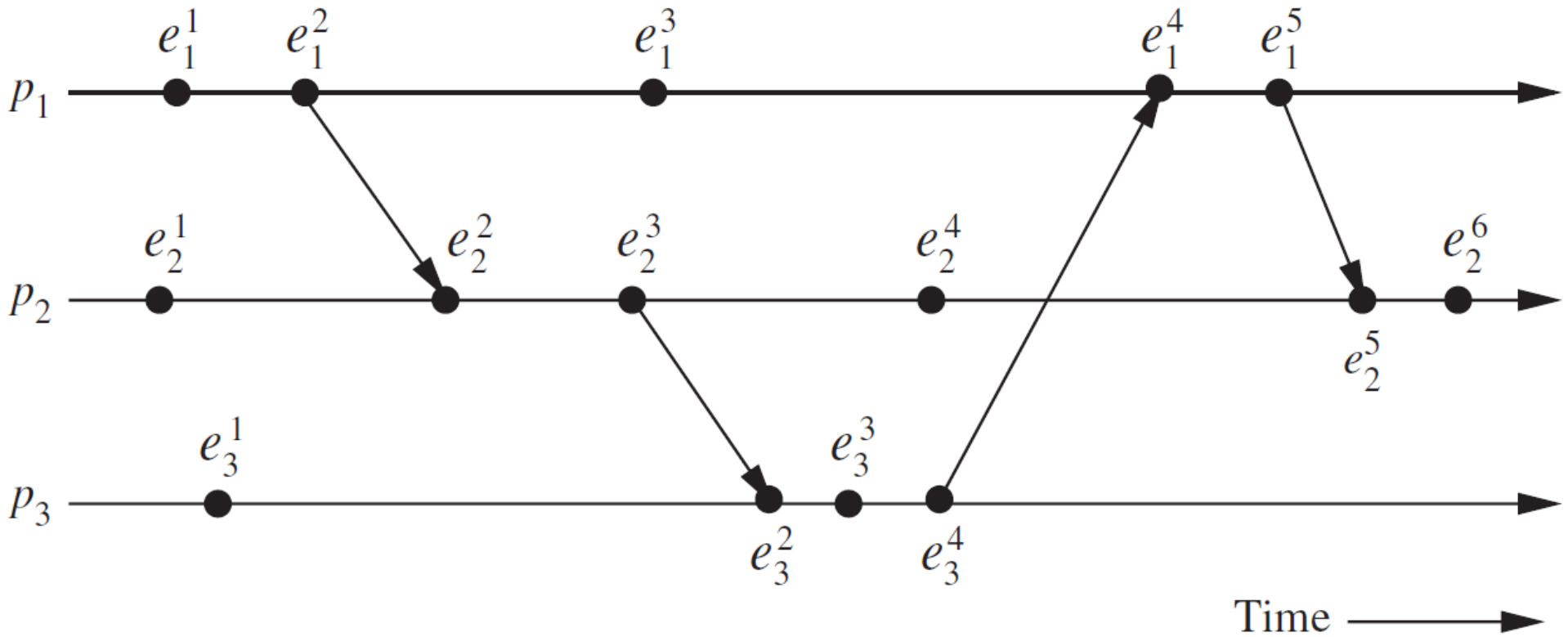
Cauzalitatea reprezintă relația dintre două (sau mai multe) evenimente în care unul are o posibilă influență asupra celorlate.

Un eveniment e^1 (localizat în P_i) poate influența cauzal evenimentul e^2 numai dacă e^1 are loc înaintea lui e^2 la P_i (fiecare nod are o execuție locală secvențială).

Procesul P_i poate influența P_j doar livrând un mesaj către P_j. De aceea, un eveniment e^1 (localizat în P_i) poate influența cauzal evenimentul e^2 din P_j numai dacă e^1 este evenimentul care trimite mesaj m de la P_i la P_j, iar e^2 este evenimentul de primire la P_j.

În al treilea caz, e^1 poate influența cauzal pe e^2 indirect prin alte evenimente cauzale.

Ordine cauzală



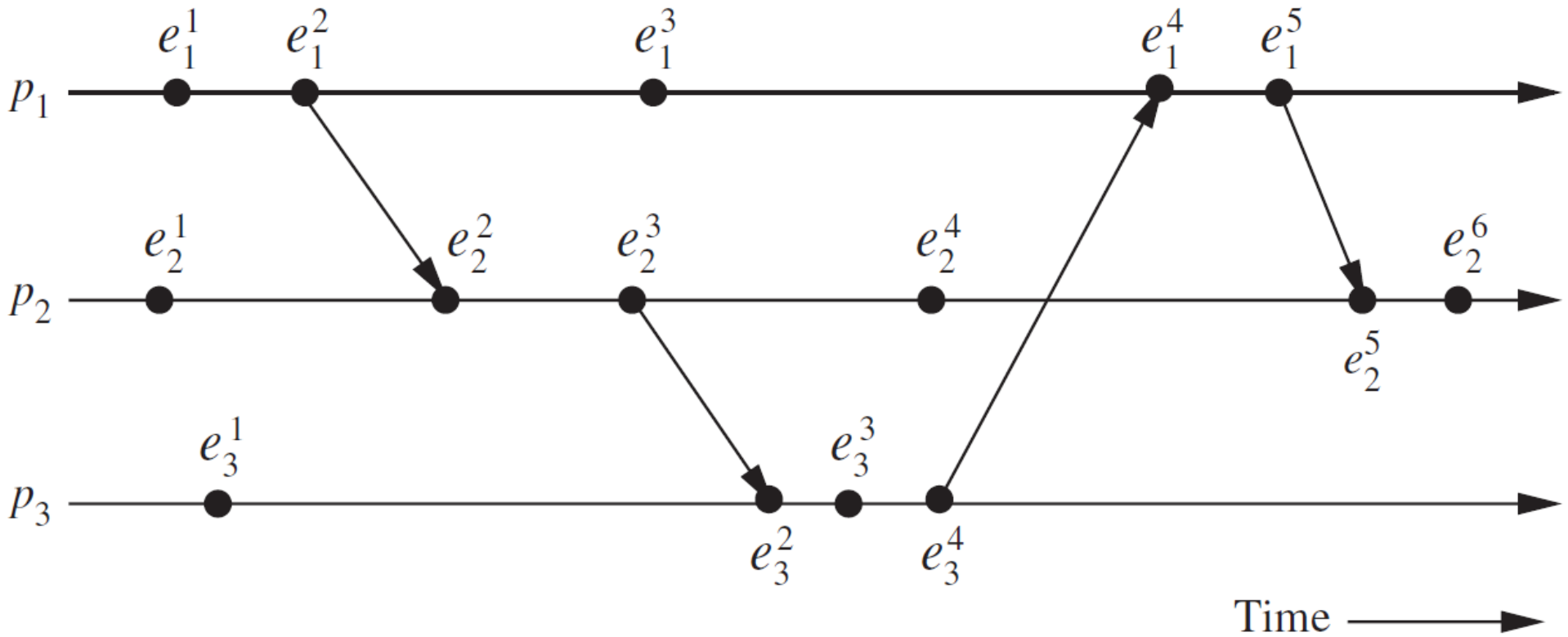
Ordine cauzală

Relația „întâmplat inainte” („*happens before*”) $<_H$ sau $\rightarrow$ între două evenimente e^1 și e^2 denotă *ordinea cauzală,* și are loc dacă unul dintre următoarele cazuri este adevărat:

1. e^1 și e^2 au loc pe același procesor și e^1 are loc înaintea lui e^2 $(e^1 \rightarrow e^2)$
2. e^1este livrarea mesajului m de P_i la P_j, iar e^2 este evenimentul de primire la P_j
3. Există e^t astfel încât $e^1 \rightarrow e^t$ *și* $e^t \rightarrow e^2$

Două evenimente sunt *concurente* $e^1||e^2$ dacă nici $e^1 \rightarrow e^2$, nici $e^2 \rightarrow e^1$nu au loc.

Ordine cauzală



Ordine cauzală

Teoremă. Fie $E = \{x^0, e^1, x^1, e^2, x^2 \ldots\}$ și V $= \{e^1, e^2, \ldots\}$. De asemenea, notăm P $= \{f^1, f^2, \ldots\}$ o permutare a lui V care păstrează ordinea cauzală, i.e. P $= \{f^1, f^2, \ldots\}$ menține ordinea cauzală în V dacă:

$$\forall(f_i, f_j), f_i \to f_j \Rightarrow i < j.$$

Atunci P_i nu poate distinge între cele două traiectorii E și F $=$ $\{C^0, f^1, C_F^1, f^2, C_F^2 \ldots\}$.

Corolar: Nu există un algoritm distribuit care observă ordinea globală a evenimentelor (i.e. diagram spațiu-timp) peste toate traiectoriile.

Ordine cauzală

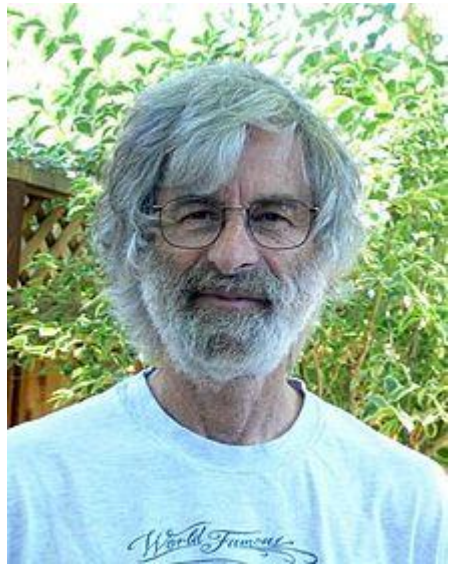
Problemă: Cum detectăm cauzalitatea locală a evenimentelor?

- Evenimentele locale ale unui singur proces sunt ordonate local.

Idee:

- Fiecare P_i păstrează o structură de date care cuprinde:
  - Un ceas logic local care măsoară progresul în P_i
  - O reprezentare a vederii lui P_i asupra ceasului global. Permitem lui P_i să atașeze un marcaj al timpului evenimentelor sale.

Ceasuri logice Lamport



Mecanism introdus de Leslie Lamport în 1978.

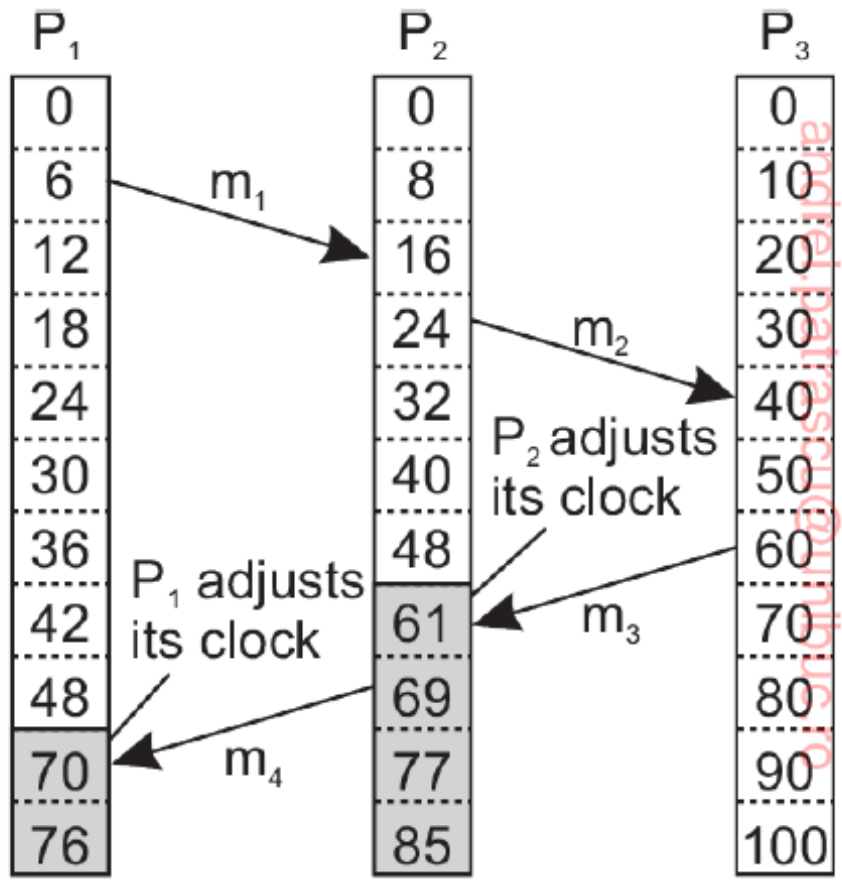
- Ceas logic = marcaj de timp C asociat unui eveniment
- Fiecare P_i întreține un ceas local C_i (scalar, care reflectă percepția locală și globală). La fiecare eveniment local (de calcul) $C_i = C_i + d \ \ (d > 0)$.
- De asemenea, la fiecare eveniment de comunicație $P_i \rightarrow_m P_j$
  - P_i atașează mesajului m valoarea curentă locală a ceasului C_i
  - P_j recepționează mesajul m și execută: $C_j := \max\{C_j, C_{msg}\}, C_j := C_j + 1$

Ceasuri logice Lamport

P_2 ajustează ceasul său local folosind timpul primit de la P_3 (increment d $= 1$)

P_1 ajustează ceasul său local folosind timpul primit de la P_2

Proprietate de consistență:
$\mathrm{A} \rightarrow B$ implică $\mathrm{C}(\mathrm{A}) < C(B)$



Ceasuri logice Lamport

Algoritm de incrementare:

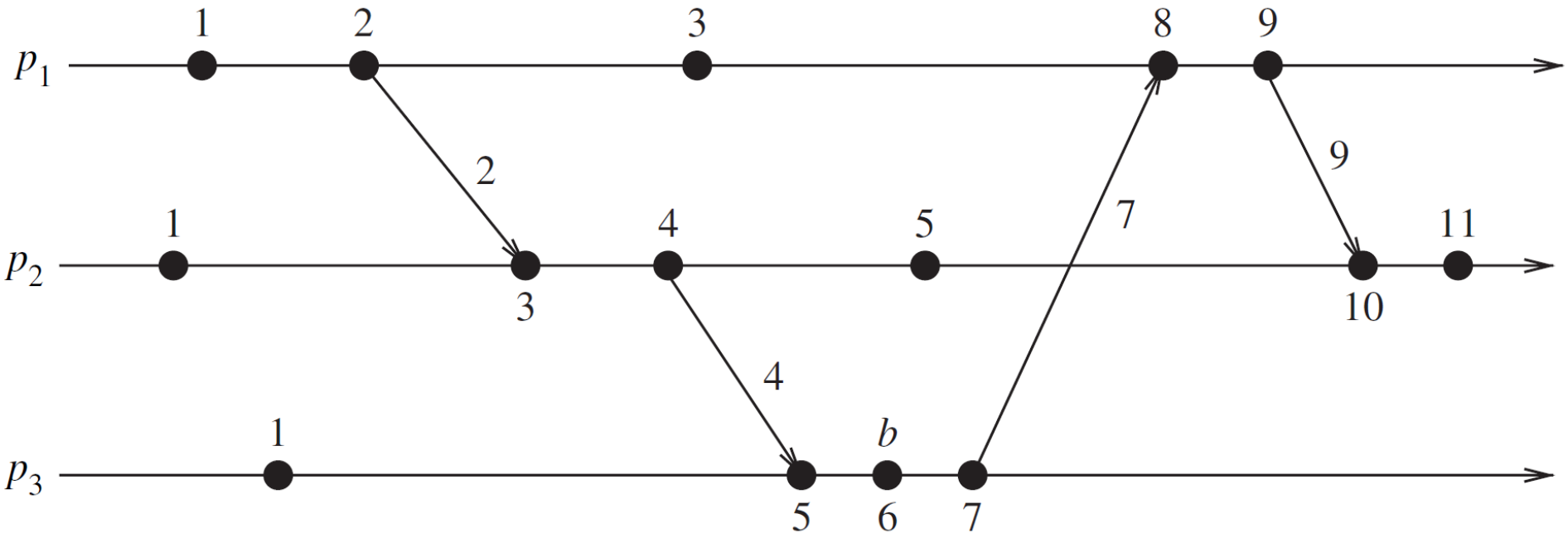
1. Înaintea execuției unei operații, P_i incrementează: $C_i = C_i + 1$.
2. Când procesul P_i livrează mesajul m către P_j, adaugă marcajul
$$ts(m) := C_i$$
3. P_j recepționează m, ajustează contorul local la:
$$C_j = \max\{C_j, ts(m)\}$$

și incrementează C_j.

Nu are loc consistența tare: $C(a) < C(b)$ nu implică a $\rightarrow b$

Actualizarea unui ceas scalar nu reține valorile de timp ale vecinilor!

Ceasuri logice Lamport



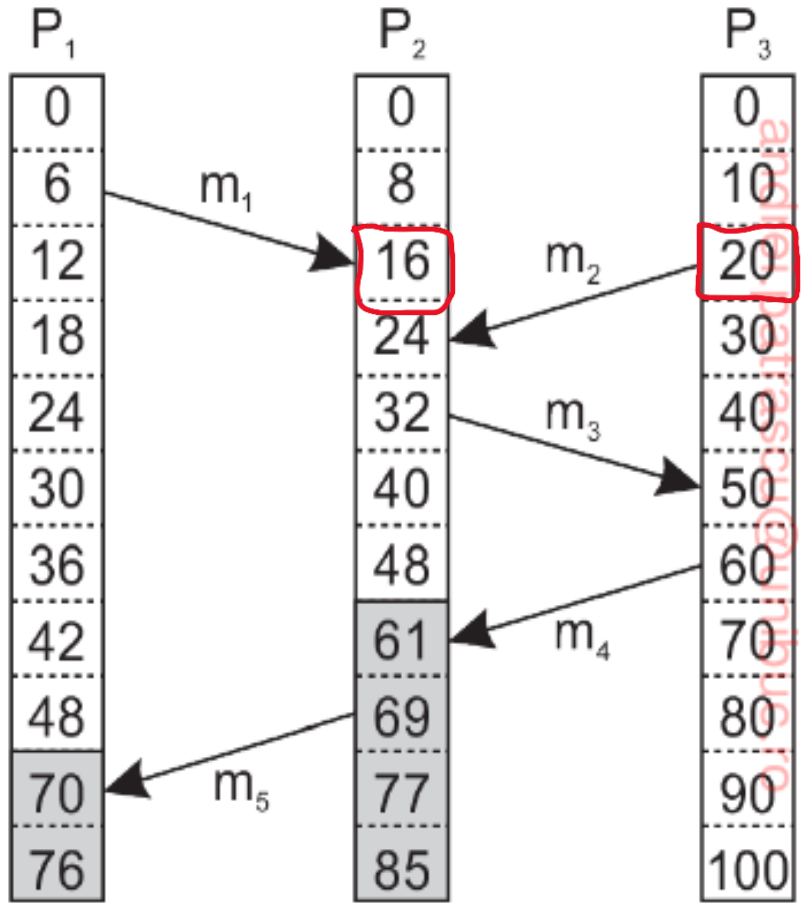
Ceasuri vectoriale

- Ceasurile logice scalare nu capturează cauzalitatea!

- Sunt necesare mai multe dimensiuni (vectori).

Printre primele referinte care au introdus ceasurile vectoriale:

Fidge, Colin J. (February 1988). "Timestamps in message-passing systems that preserve the partial ordering" (PDF). In K. Raymond (ed.). *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)*. Vol. 10. pp. 56–66. Retrieved 2009-02-13.



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: $V_i[i] = V_i[i] + 1$

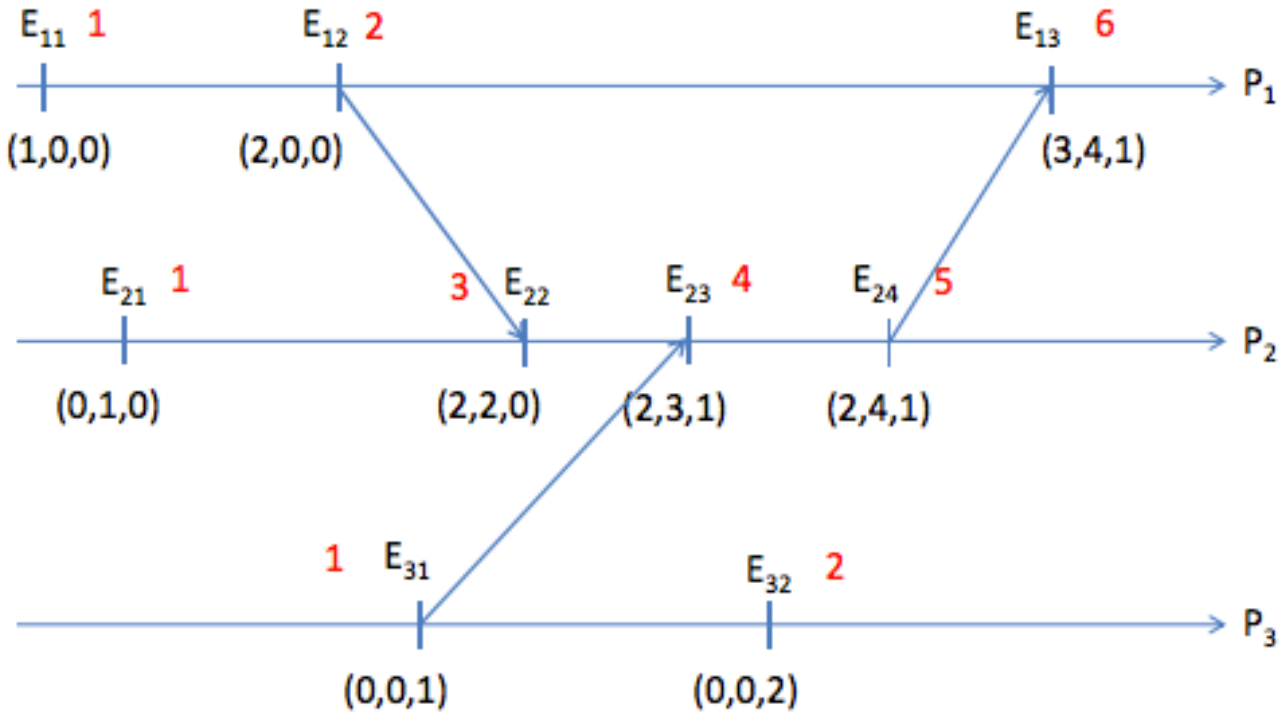
Când m este livrat de P_i la P_j atașează V_i la mesajul m

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

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

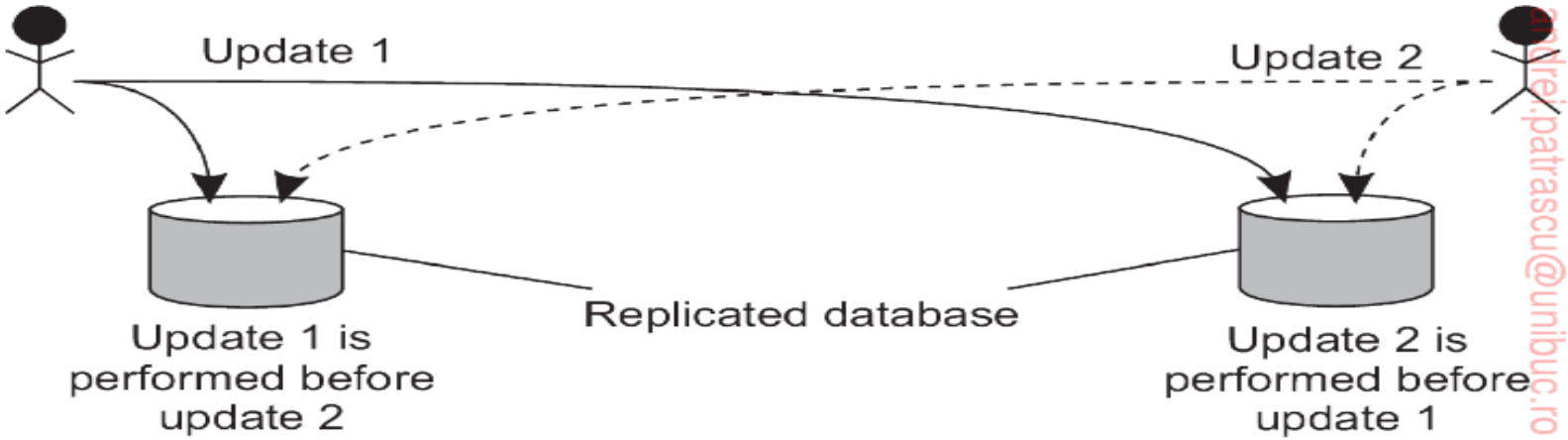
Ceasuri vectoriale

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 $V(B)! < V(A)$

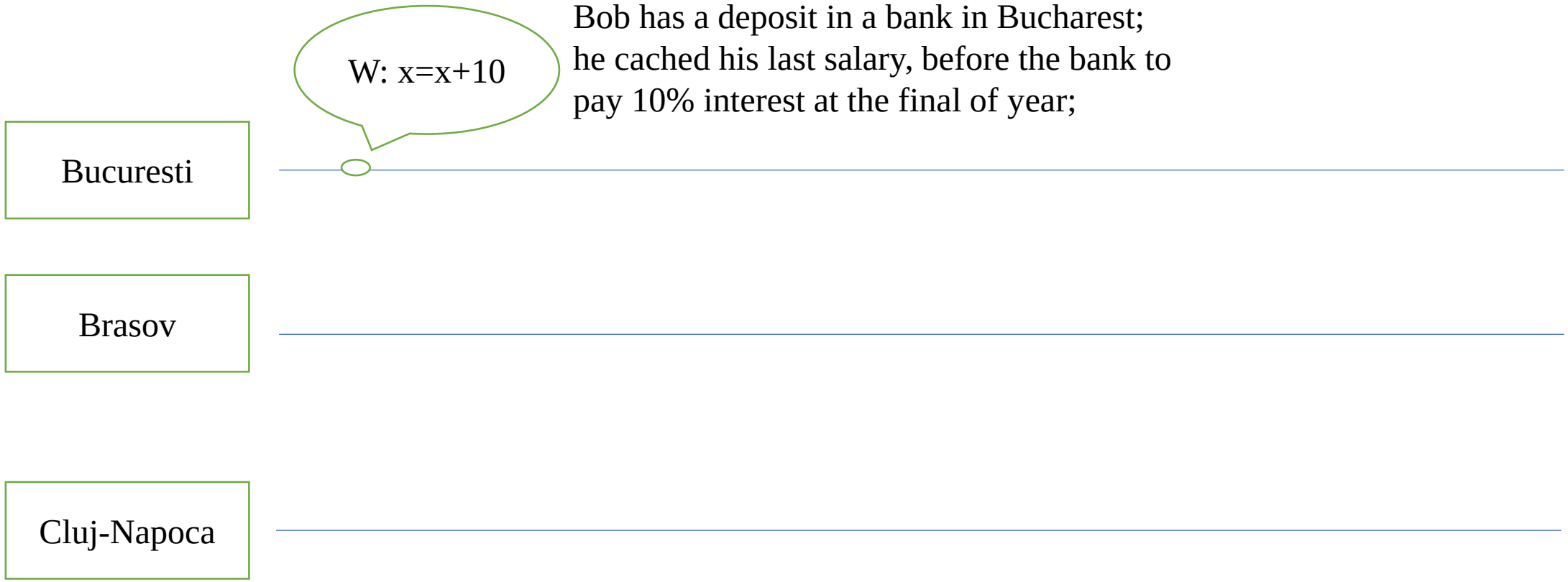


Aplicații

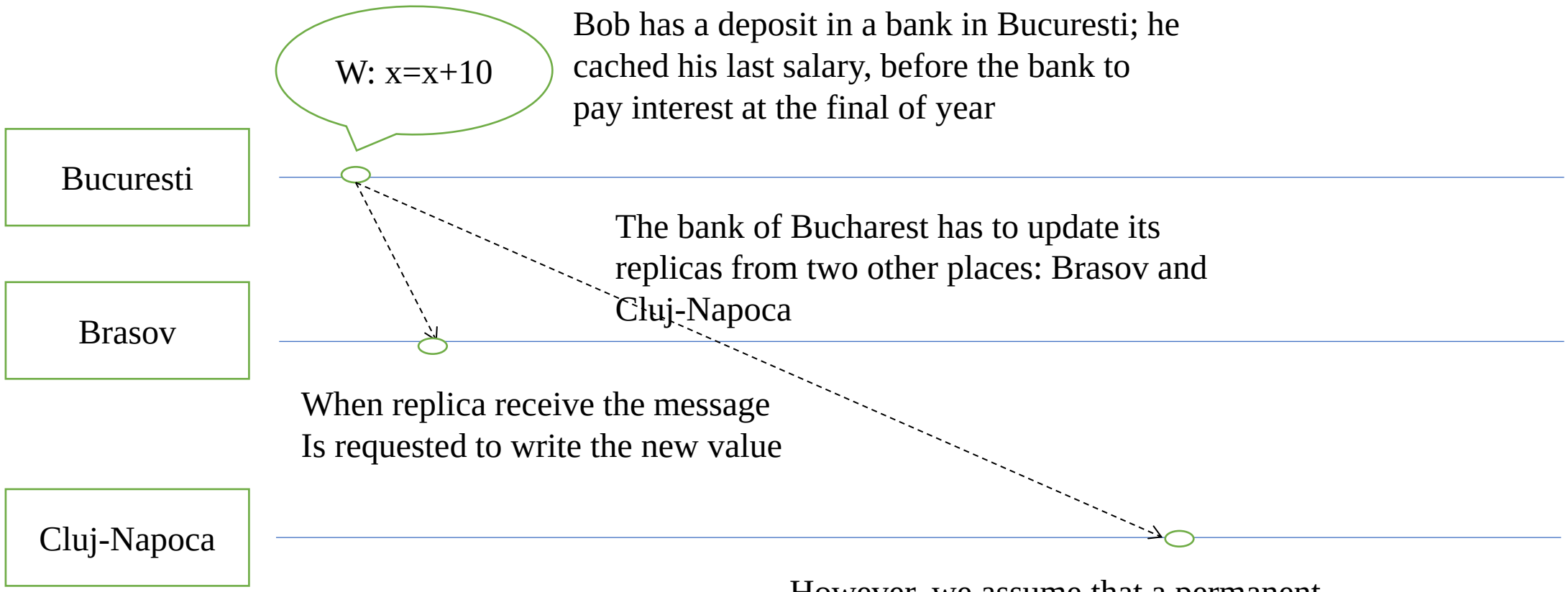
- Consistenta baze de date (e.g. Amazon Dynamo)
- Rezolvare conflicte
- Sisteme bancare



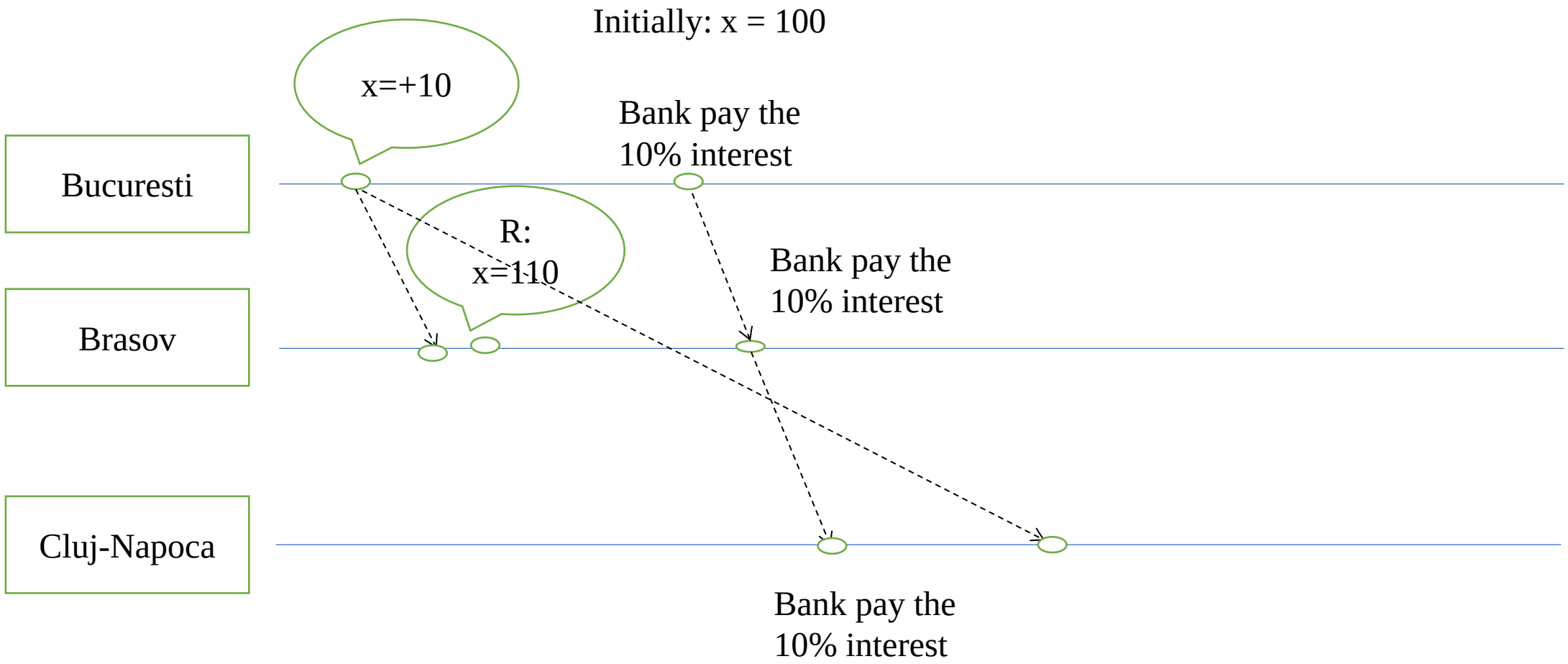
Consistență cauzală



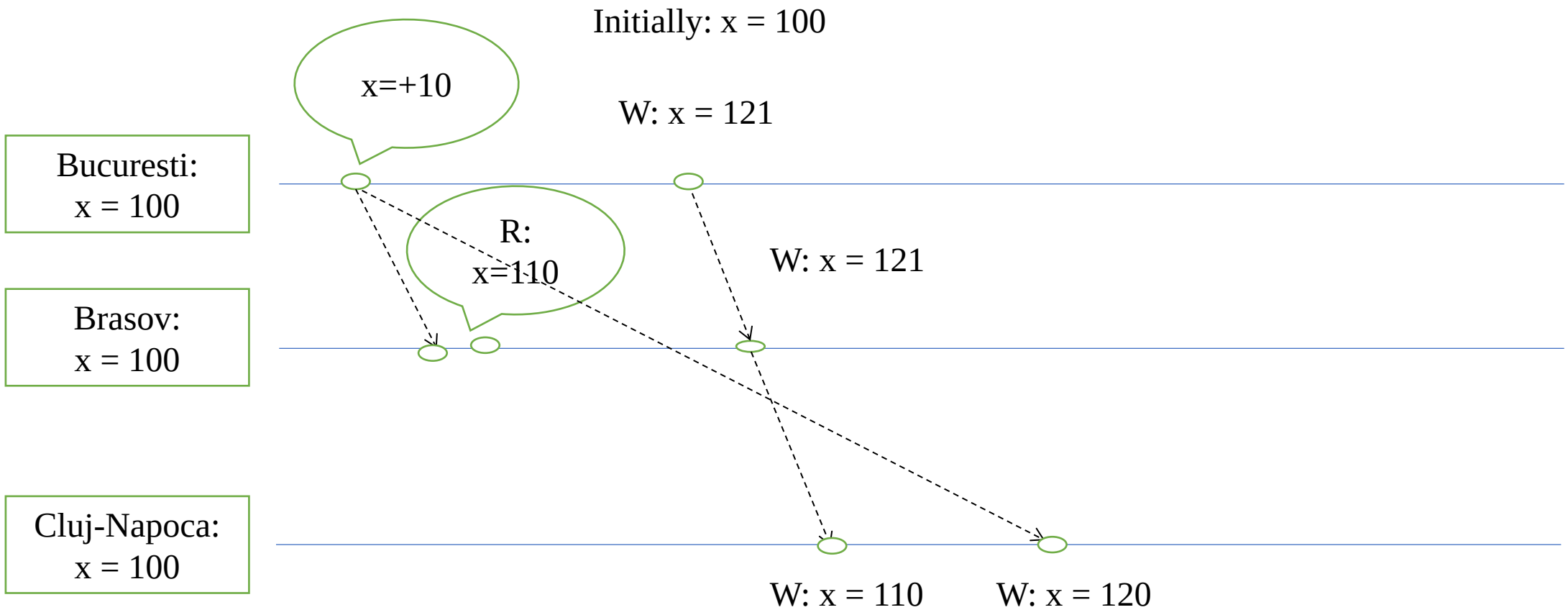
Consistență cauzală



Consistență cauzală

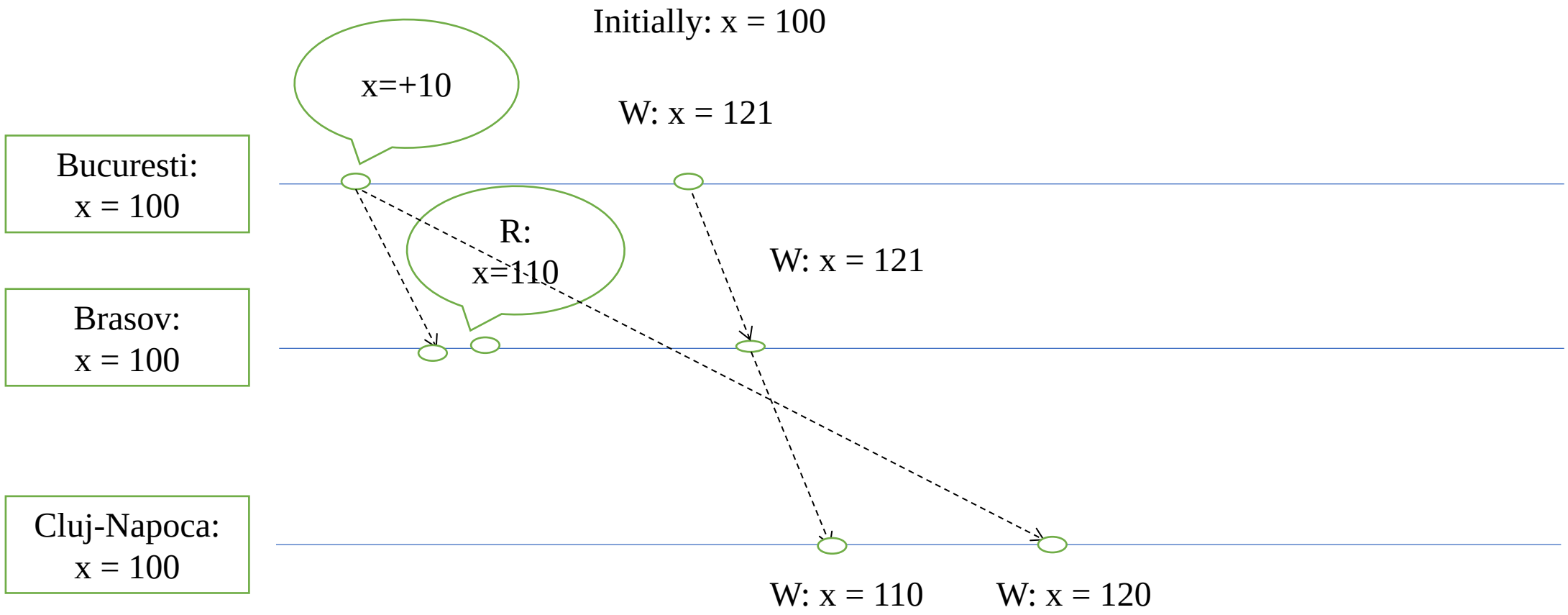


Consistență cauzală



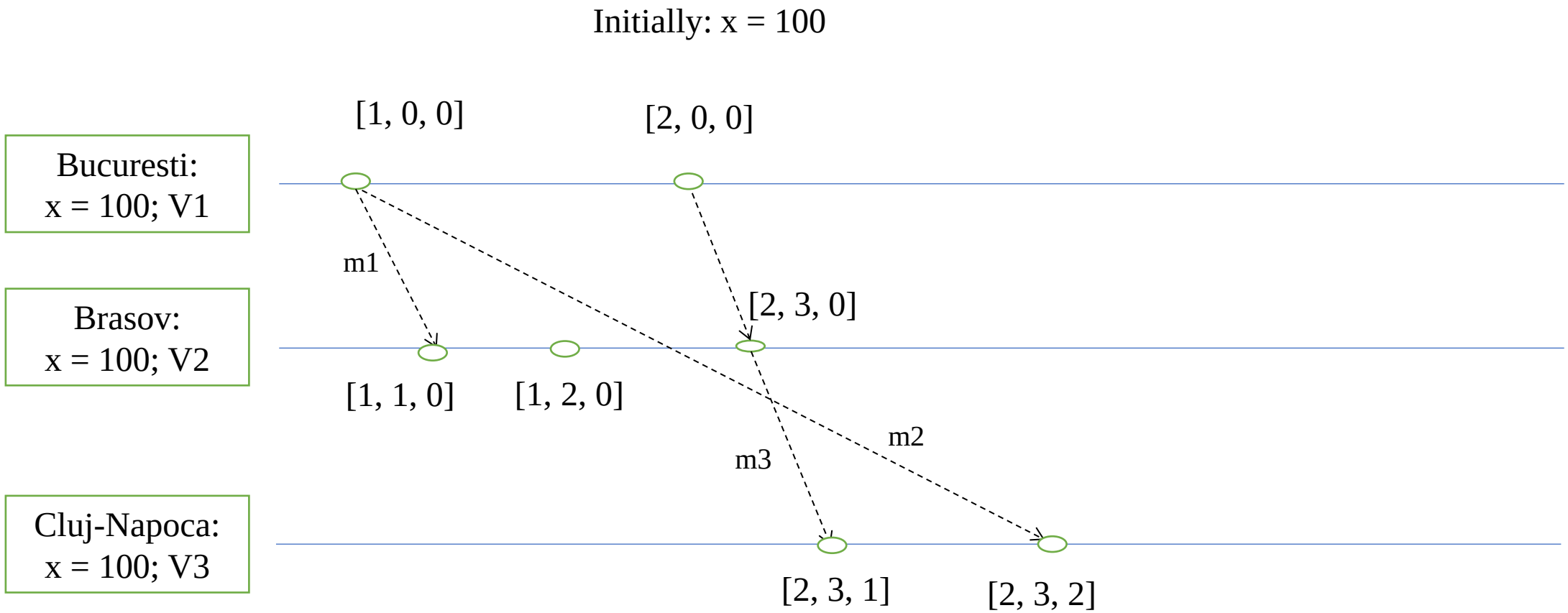
In third replica, the final sum results in 120 since it attempts to realize the write operations in reversed order; What to do?

Consistență cauzală



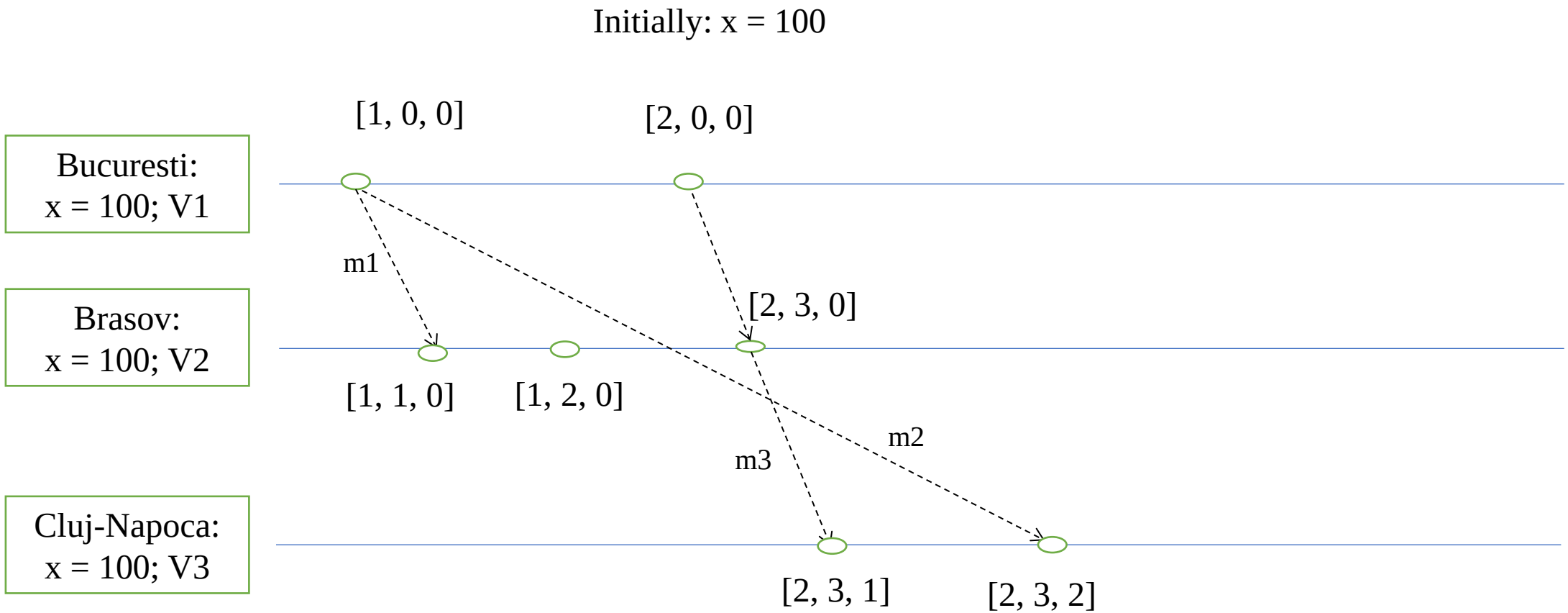
In third replica, the final sum results in 120 since it attempts to realize the write operations in reversed order; What to do? **Vector clocks**

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

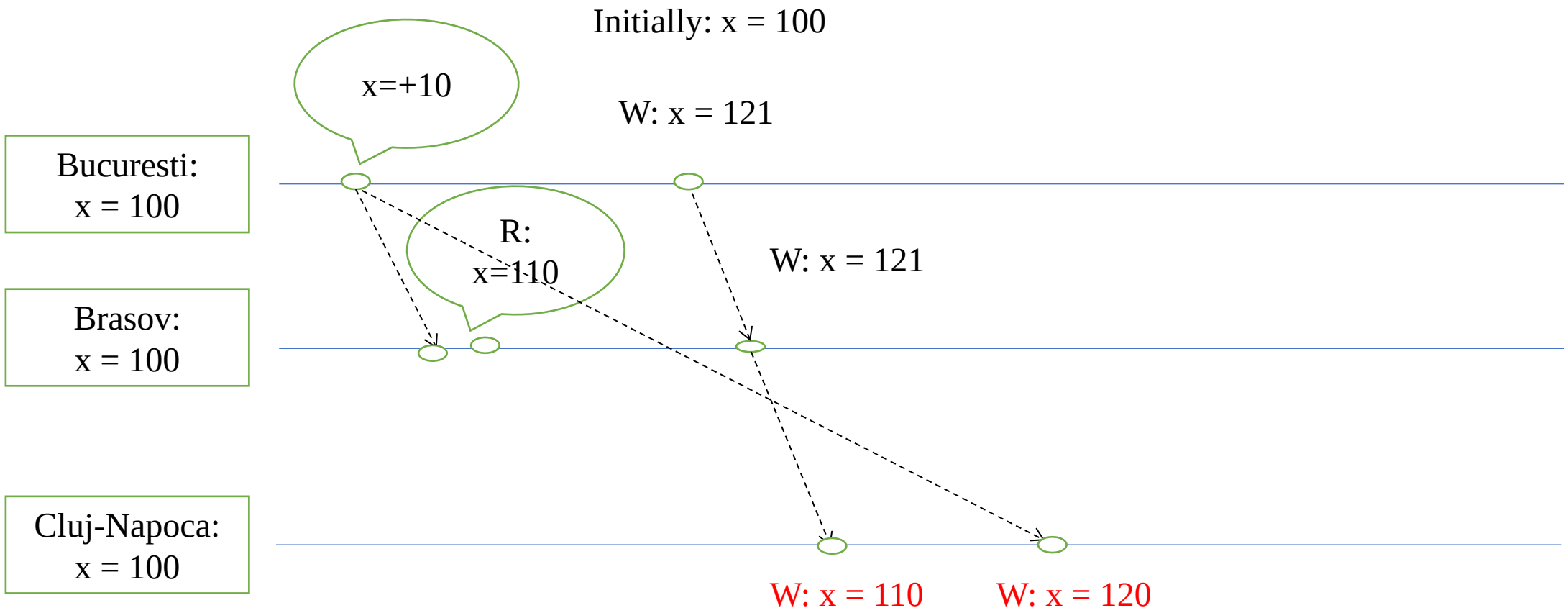


Timestamps: ts(m1) = [1, 0, 0]; ts(m2) = [1, 0, 0]; ts(m3) = [2, 3, 0]
Vector clocks: V1(send(m2)) < V2(send(m3)), therefore

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

Ceasuri vectoriale

Câteva dezavantaje:

- Dimensiunea mesajelor la fiecare iterație este egală cu numărul de noduri total
- Dacă CV este atașat unui obiect cu mai multe câmpuri, atunci modificarea unui singur câmp necesită actualizarea CV la nivel de obiect și pasarea întregului obiect între procese.
- Conflictele non-cauzale nu pot fi rezolvate cu CV
- În general, CV doar indică apariția unui conflict, nu și modul de rezolvare.