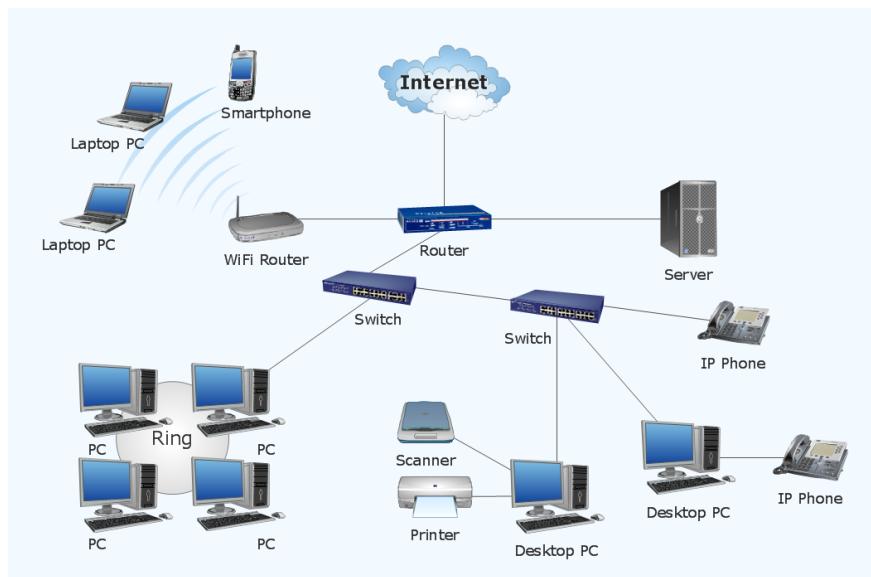


Sisteme și algoritmi distribuiți

Curs 1

FMI – UNIBUC
2024 - 2025



Organizare

Curs 2h/saptamana: A. P.

Laborator 2h/saptamana: Marius Mihailescu

Evaluare:

1. Examen scris (50%)
2. Teme laborator (50%)

Promovare: nota 5 examen + nota 5 laborator.

Materiale și comunicare: platforma Teams.

Adresa e-mail: andrei.patrascu@fmi.unibuc.ro

Plan materie

1. Introducere. Modele și arhitecturi SD.

2. Comunicație în SD. Ceasuri logice.

3. Sisteme sincrone

3.1 Algoritmi OSD: alegere lider, consens, sincronizare

3.2 Defecte

3.3 Consens și alte probleme numerice: medie, evaluare funcții, sisteme liniare

3.4 Toleranță la defecte

4. Sisteme asincrone

4.1 Organizarea SDa: teorema imposibilitate, consens relaxat

4.2 Medie și sisteme liniare în context asincron

4.3 Toleranță la defecte

5. Algoritmi distribuiți aleatori:

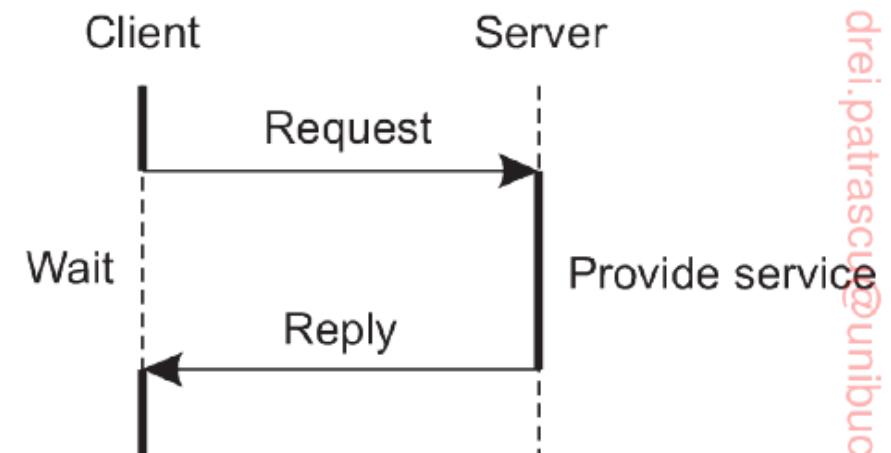
5.1 Paradigma gossip

5.2 Consens și alte probleme

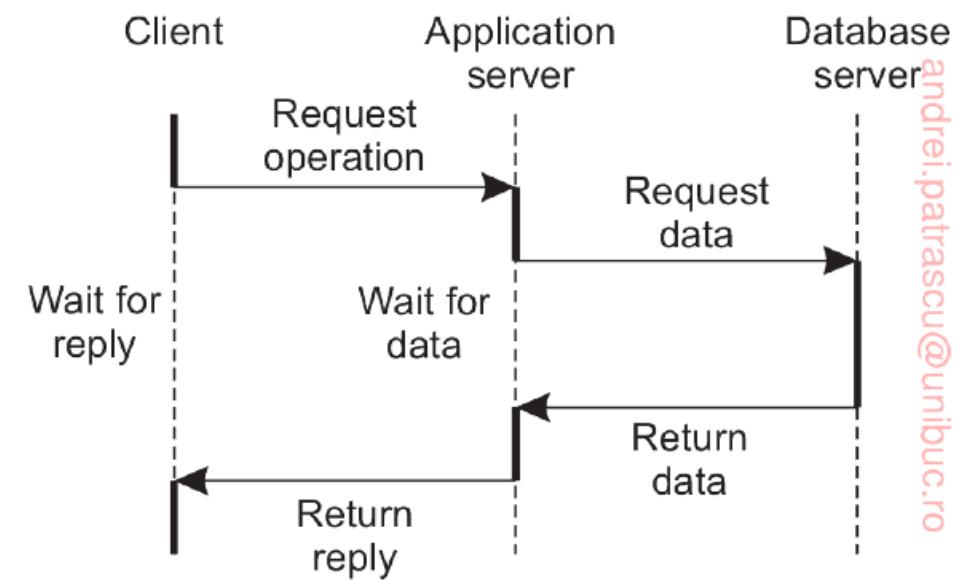
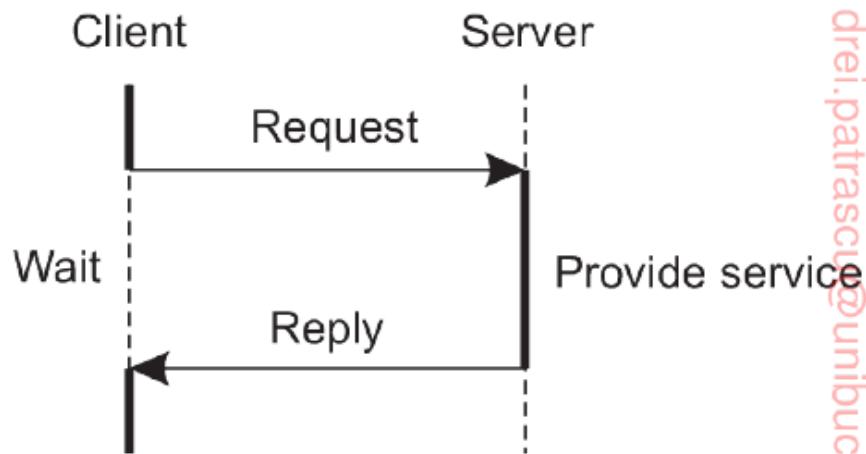
5.3 Toleranță la defecte

Client-Server

- Un nod/proces are calitatea de client sau server
- **Server** = entitate care ofera un serviciu
- **Client** = entitate care apeleaza un serviciu

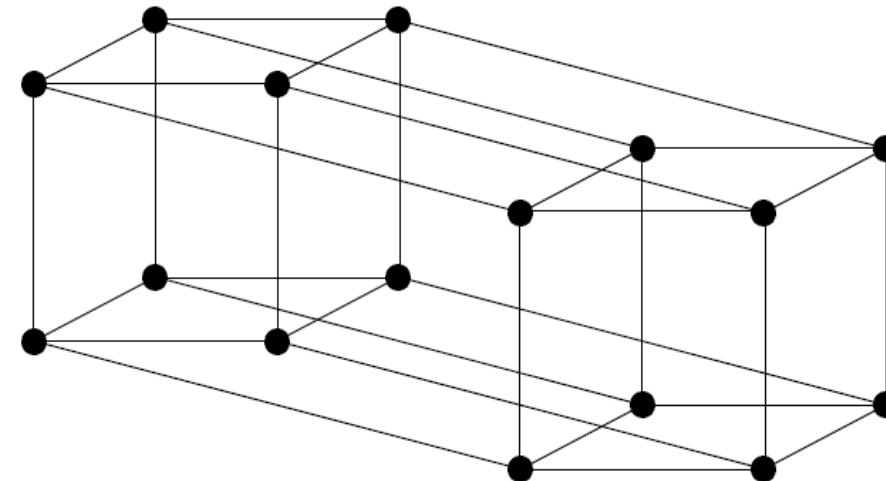
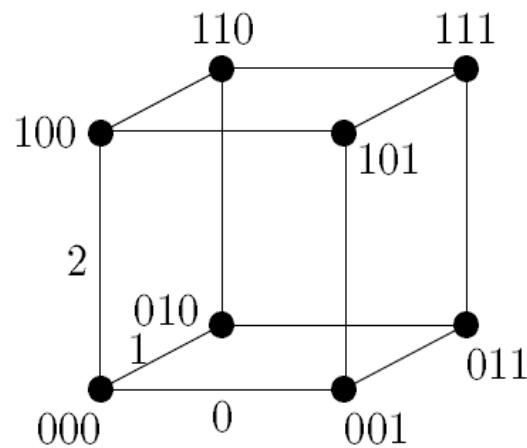


Client-Server



Sistem multiprocesor structurat

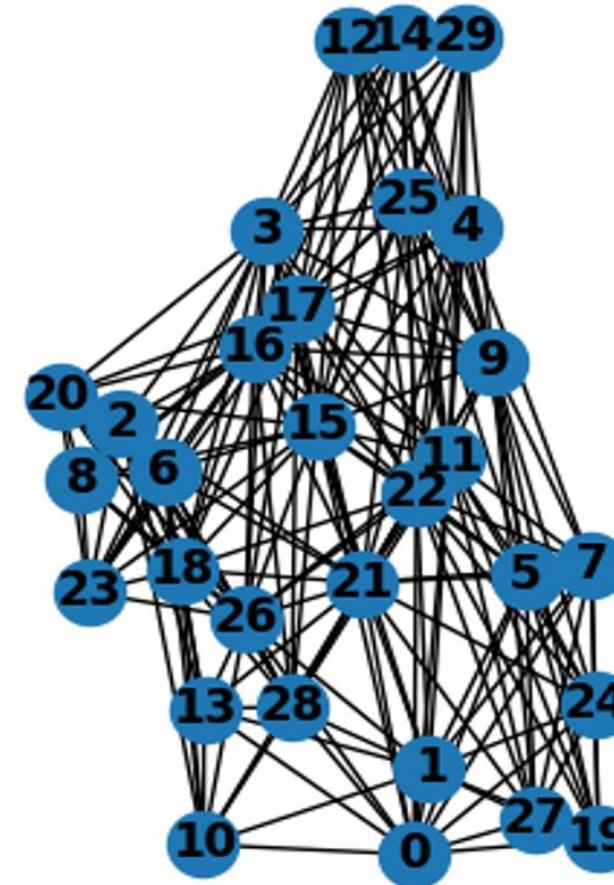
- Fiecare nod are aceleasi capacitatii
- Comunicatie sigura intre procese
- Identifier unic per proces
- Folosită în “cluster (parallel) computing”



Sisteme nestructurate (peer-to-peer)

Exemplu: Cloud Computing, Sisteme de stocare

- Specifice modelelor generale de “**sisteme distribuite**”
- Nodurile nu au un identificator unic global
- Comunicatia intre doua noduri se realizeaza prin muchiile disponibile
- Topologia potential variabila in timp
- Posibile defecte pe legaturi sau noduri



Sistem distribuit

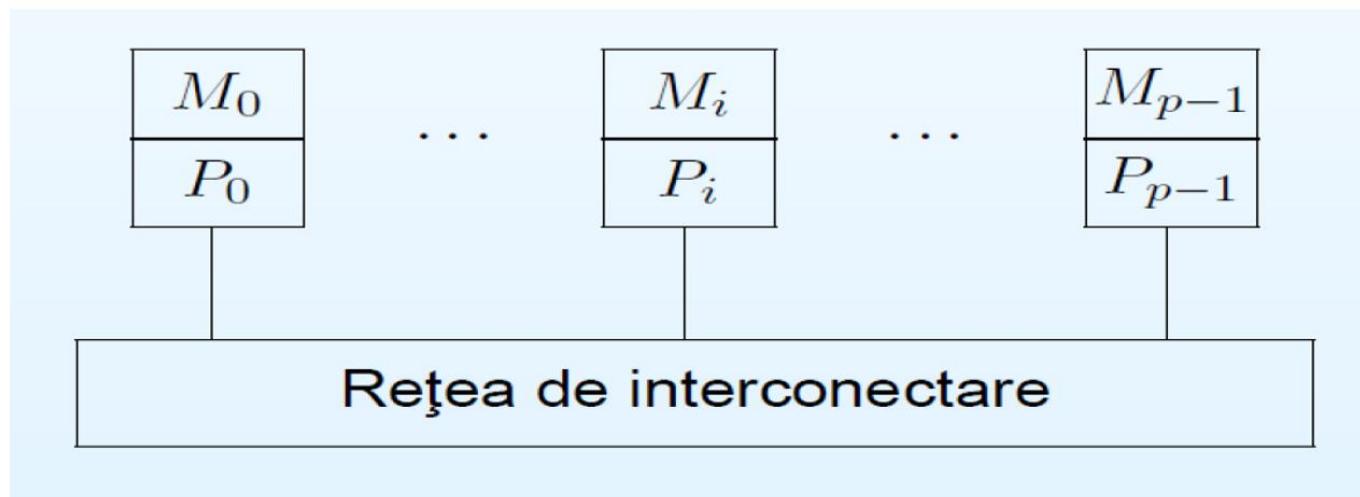
Sistem distribuit = o colecție de procese autonome care comunică peste o rețea (topologie) cu următoarele proprietăți:

- **Fiecare nod are o „vedere” locală asupra sistemului.** Un nod al sistemului cunoaște și comunică cu propria vecinătate, neavând acces la informații globale. În general, există o separare geografică a nodurilor.
- **Nu există un ceas fizic comun.** Acestui aspect se datorează caracterul “distribuit” al sistemului și este cel care cauzează lipsa sincronizării între noduri.
- **Nu există memorie partajată.** Proprietate care aduce necesitatea comunicației prin mesaje (în absența unui ceas global).
- **Autonomia și eterogenitatea nodurilor.** Noduri sunt “slab cuplate”, au viteze diferite de execuție, au sisteme de operare diferite.

Sistem distribuit vs. Sistem paralel

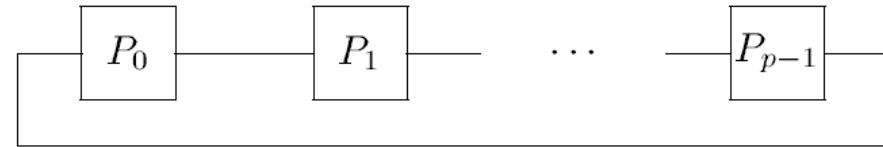
Sistem MIMD cu memorie distribuită

- ▶ Fiecare procesor are memorie proprie (arhitectura locală cu RISC și memorie ierarhică, de obicei)
- ▶ Comunicația se face printr-o rețea de comunicație, prin mesaje explicate
- ▶ Operații favorizate: paralele, la nivel de bloc
- ▶ Comunicația prin mesaje necesită algoritmi dedicați

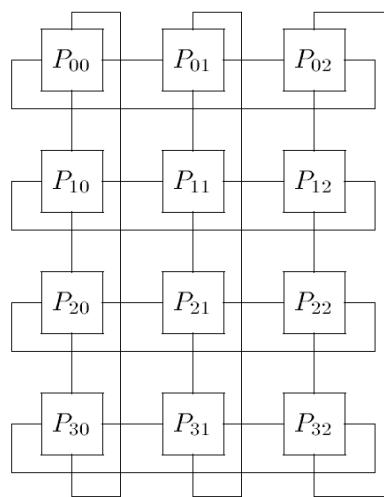
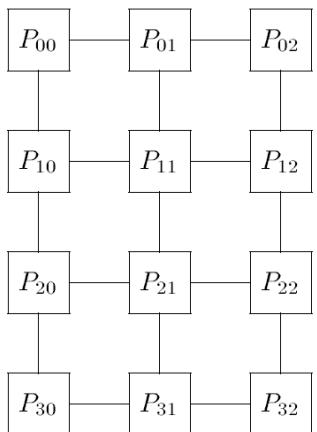


MIMD cu memorie distribuită

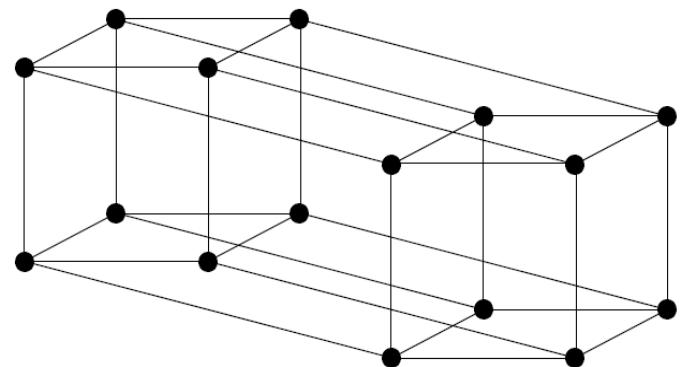
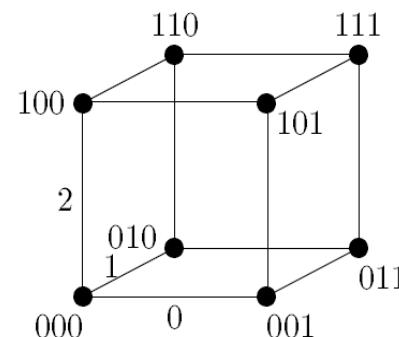
Inel



Grila/Tor



Hipercub



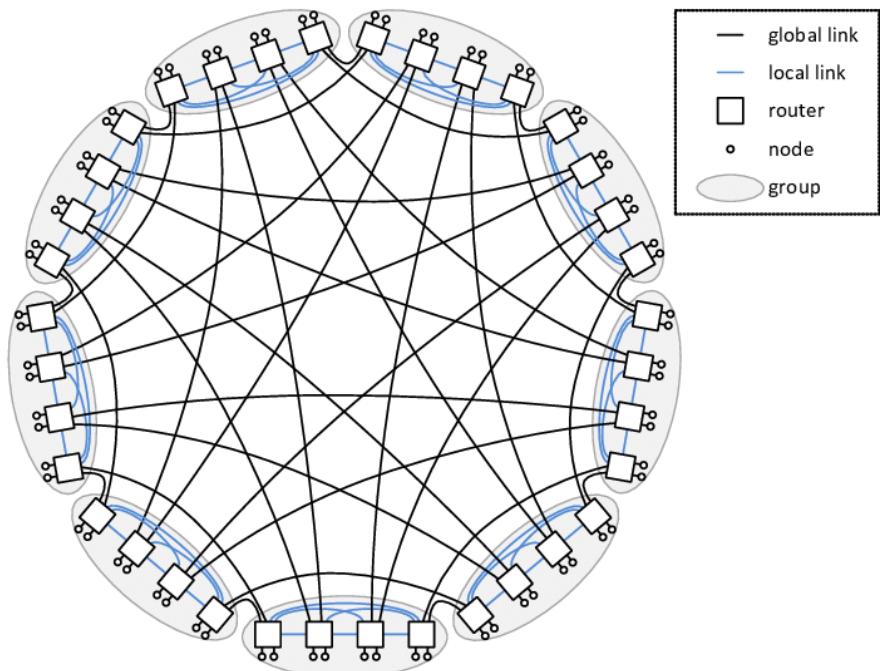
Frontier



Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	Aurora - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

Frontier cluster

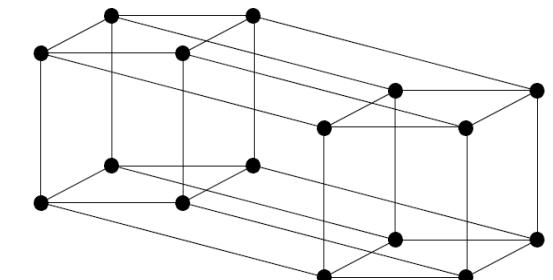
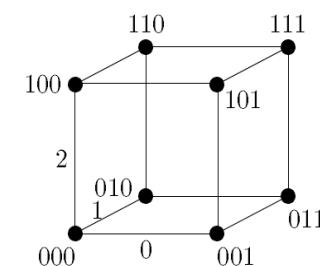
- 9.472 CPU AMD Epyc 7713 (64 cores)
 - 606.208 core-uri
- 74 rack-cabinets
- 1 rack-cabinet are 64 server-blades
- 1 server-blade are 2 noduri
- Nodurile sunt structurate pe grupuri; grupurile sunt conectate intr-o topologie “dragonfly”



Sistem paralel (Cluster computing)

- **Informații globale disponibile:** număr de noduri ale rețelei, topologia rețelei, distribuția datelor în rețea, indexarea globală a nodurilor
 - Control asupra distribuției datelor
 - Control asupra execuției locale per nod
 - Control asupra implicării nodurilor în rețea
- **Timp de comunicație** inter-noduri neglijabil/mărginit (apropiere geografică)
 - Sincronizare: ceas fizic comun

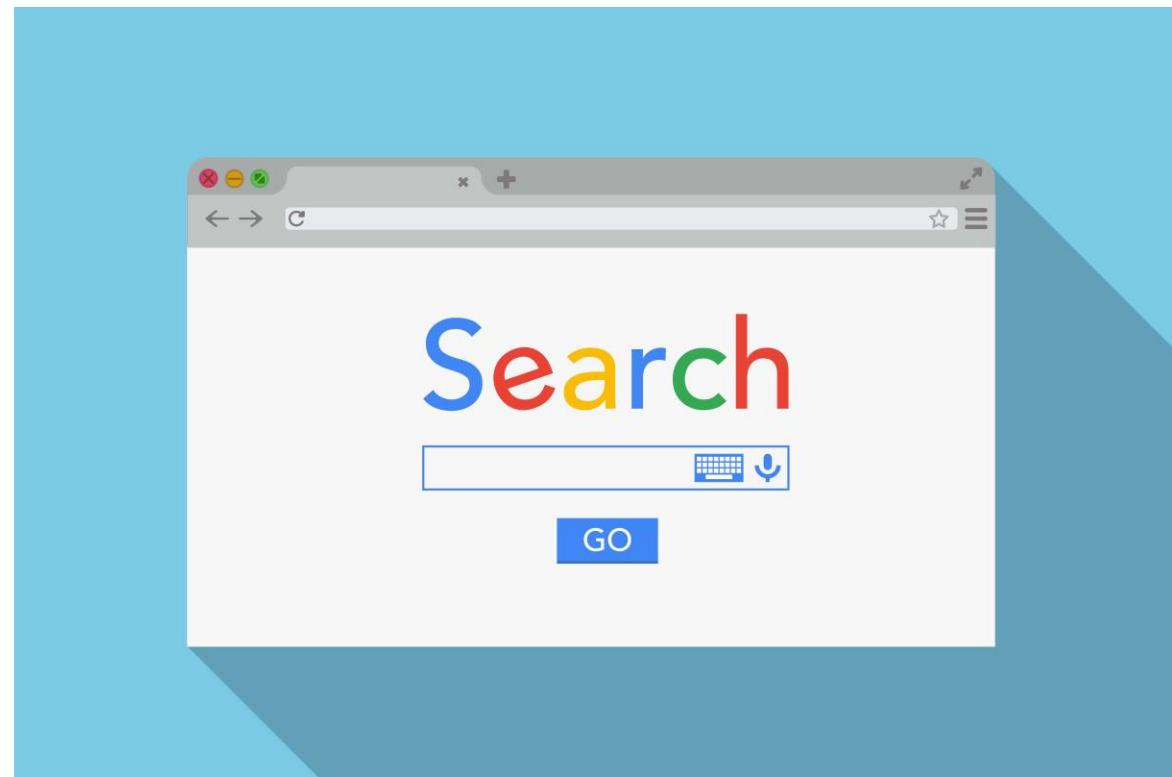
- Topologie statică
- Probabilitatea scăzută a defectelor
- Complexitatea timp vs. complexitatea mesaj



Exemple (clasificare software)

Motoare de căutare

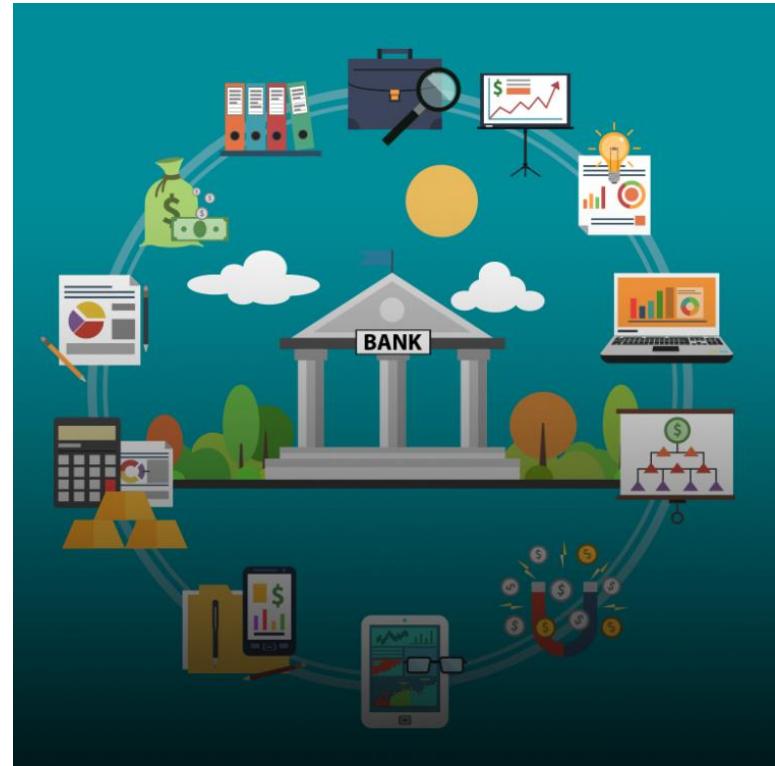
- Alg. PageRank
- Combină paradigmile anterioare



Exemple (clasificare software)

Sistem bancare

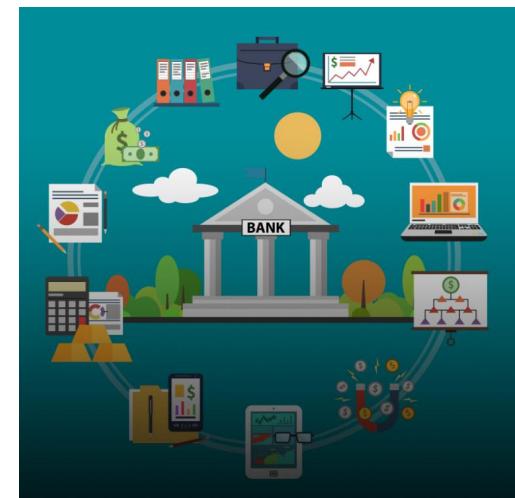
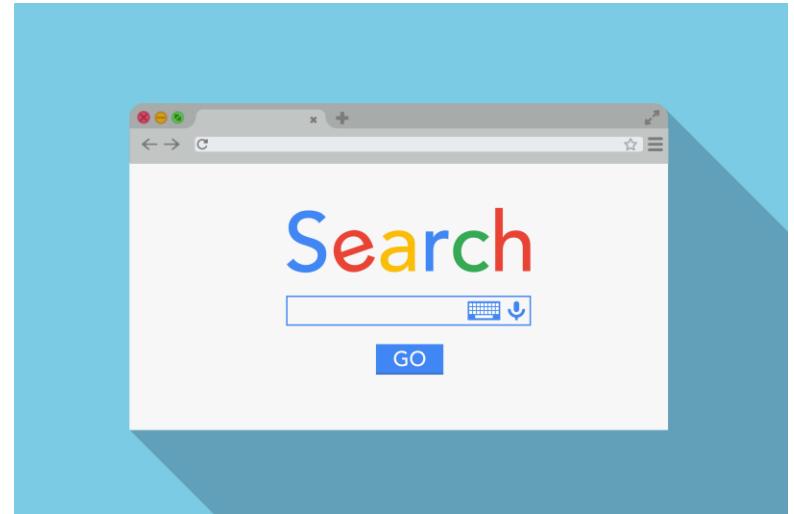
- Sistem informațional
- Nod = replică baze de date
- Probleme de consistență etc.



Exemple (clasificare software)

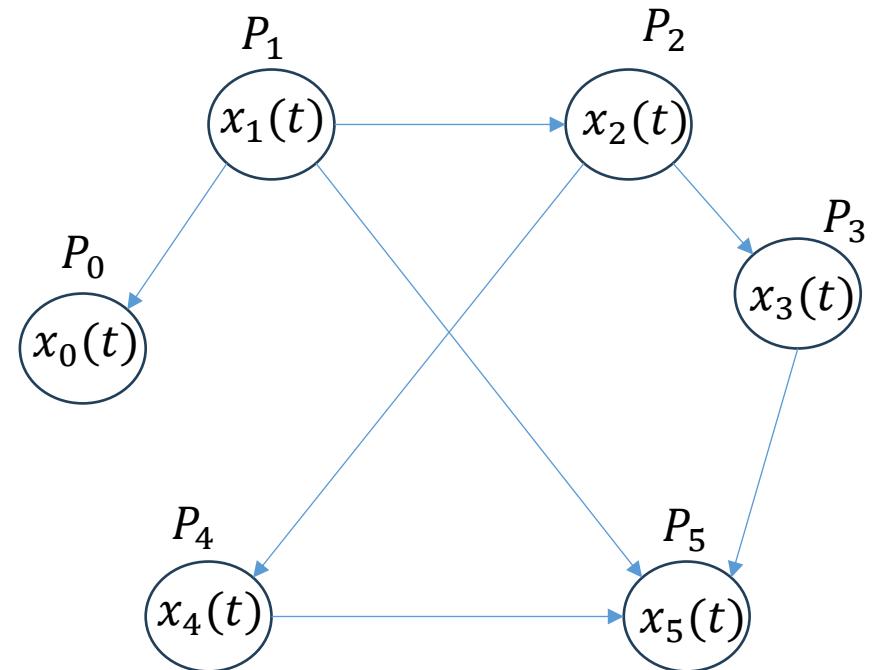
- **Distributed Computing Systems**
 - Folosite pentru calcul de înaltă performanță
 - Sisteme Cluster - Cloud

- **Distributed Information Systems**
 - Integrare funcții de business
 - Procesare tranzacții

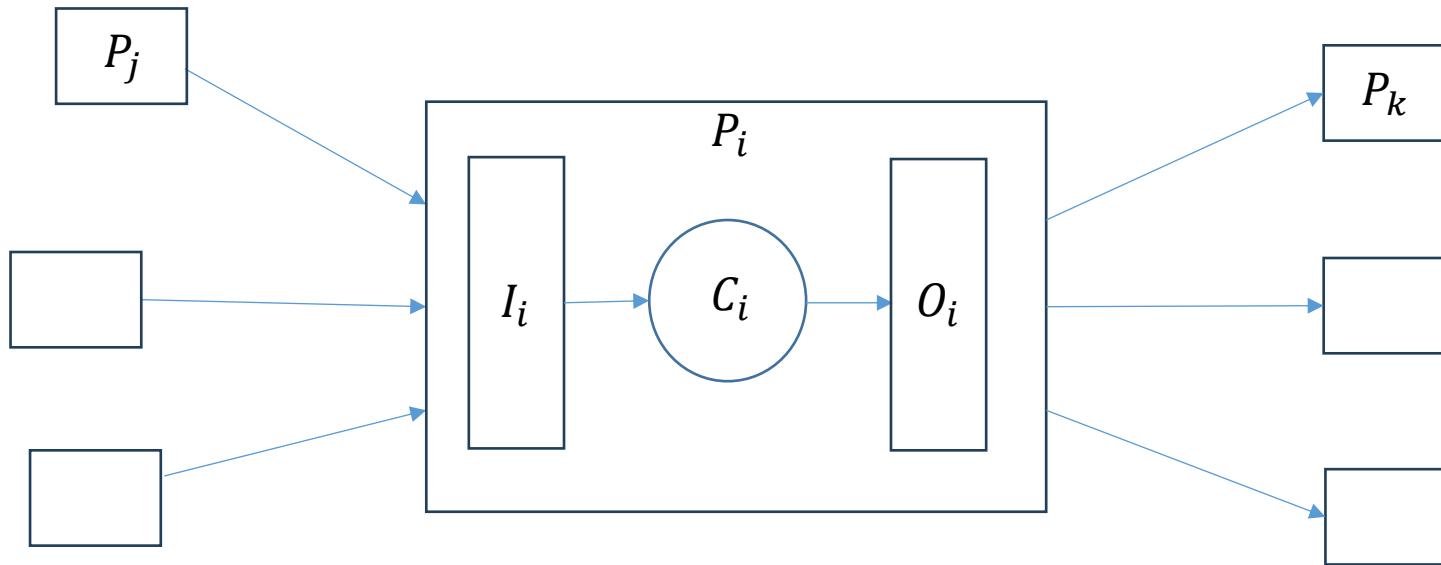


Model SD

- Noduri: P_0, P_1, \dots, P_{n-1}
- Rețea de comunicație $G = (V, E)$
 - $V = \{P_0, P_1, \dots, P_{n-1}\}$
 - $(i, j) \in E$ dacă există muchie între nodurile P_i și P_j
- Starea nodului P_i se exprima $x_i: \mathbb{N} \rightarrow \mathcal{D}$
- Stare nod P_i la momentul de timp t : $x_i(t)$
 - Temperatură
 - Locație-Viteză
 - Vot-Opinie

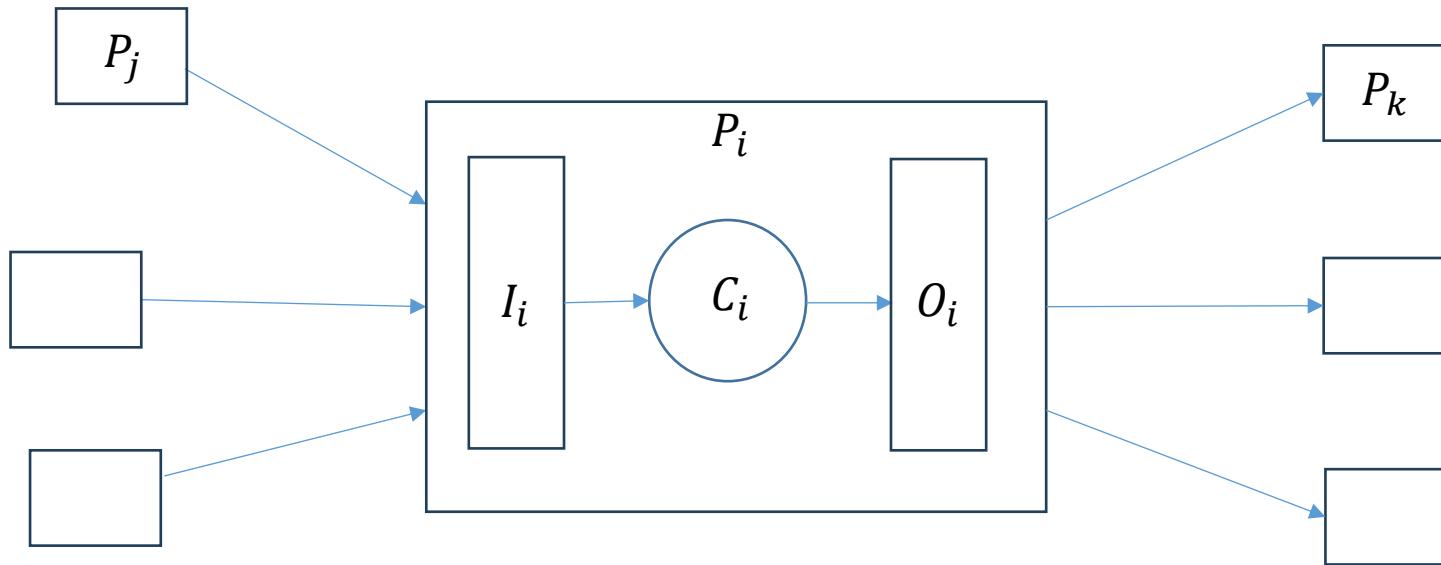


Model Proces



- Un proces primește informație la intrare (mesaje de la vecini), stocată în I_i
- Calculează noua stare pe baza info de input; în plus, depune în O_i mesaje pentru transmitere
- Transmite mesajele din O_i către vecinii de ieșire
- Reprezentăm transferul de mesaje ca evenimente separate reușite/eșuate

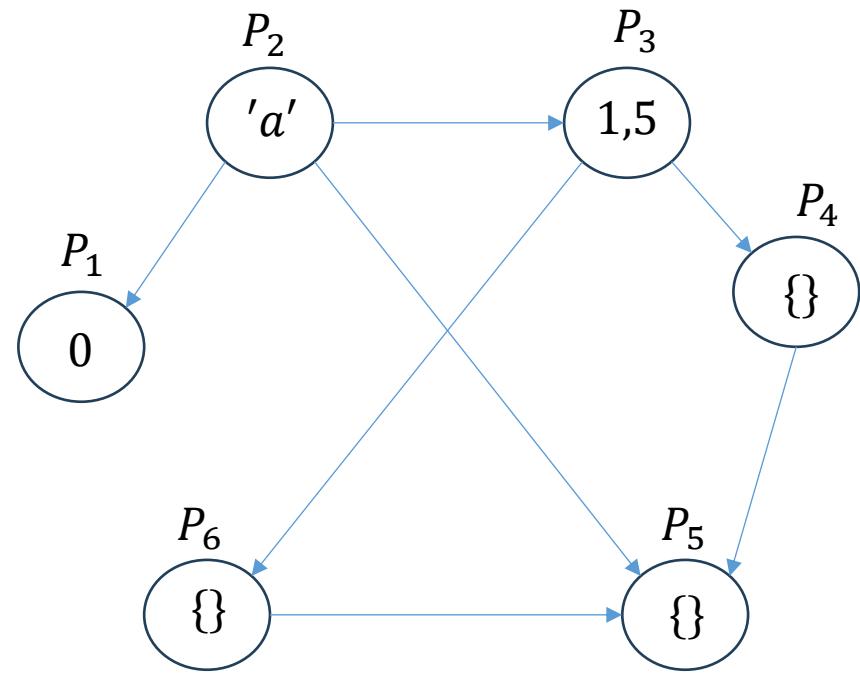
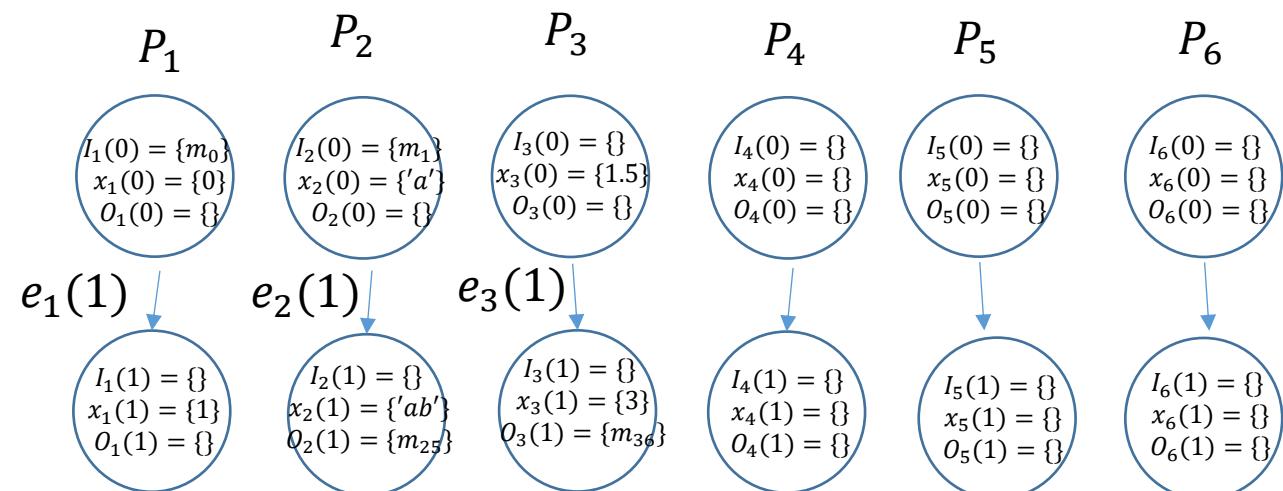
Model Proces



Mai formal:

- $I_i(t), O_i(t), x_i(t)$ buffer-ele de intrare și ieșire, și starea la momentul t
- Nodul i primește mesajul m la intrare: $I_i(t+1) = I_i(t)/\{m\}$
- Calculează noua stare pe baza intrării: $f_i(x_i(t), m) \rightarrow (x_i(t+1), \{m_1, \dots, m_k\})$
- Transmite mesajele de ieșire: $O_i(t+1) = O_i(t) \cup \{m_1, \dots, m_k\}$
- Considerăm I/O evenimente separate în rețele supuse la defecte

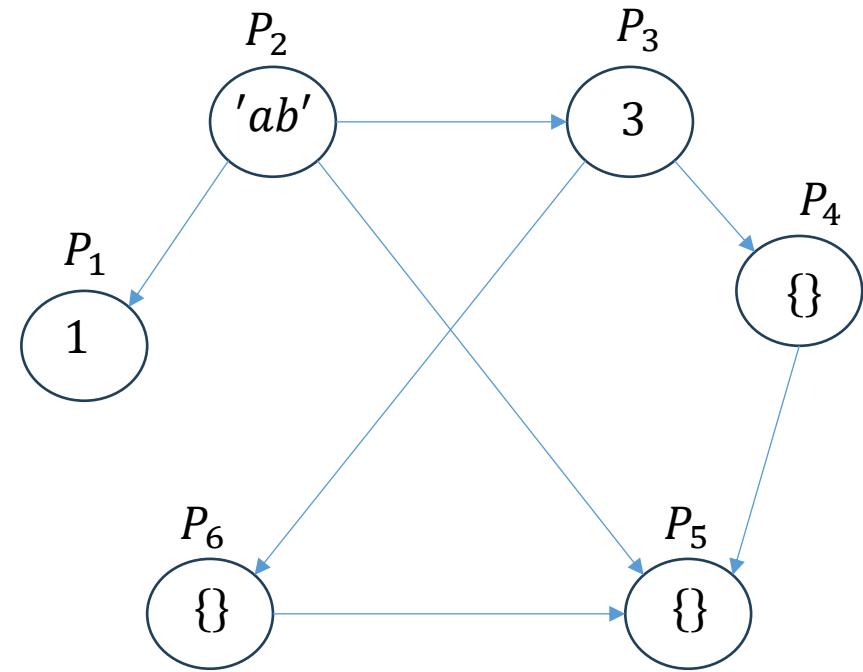
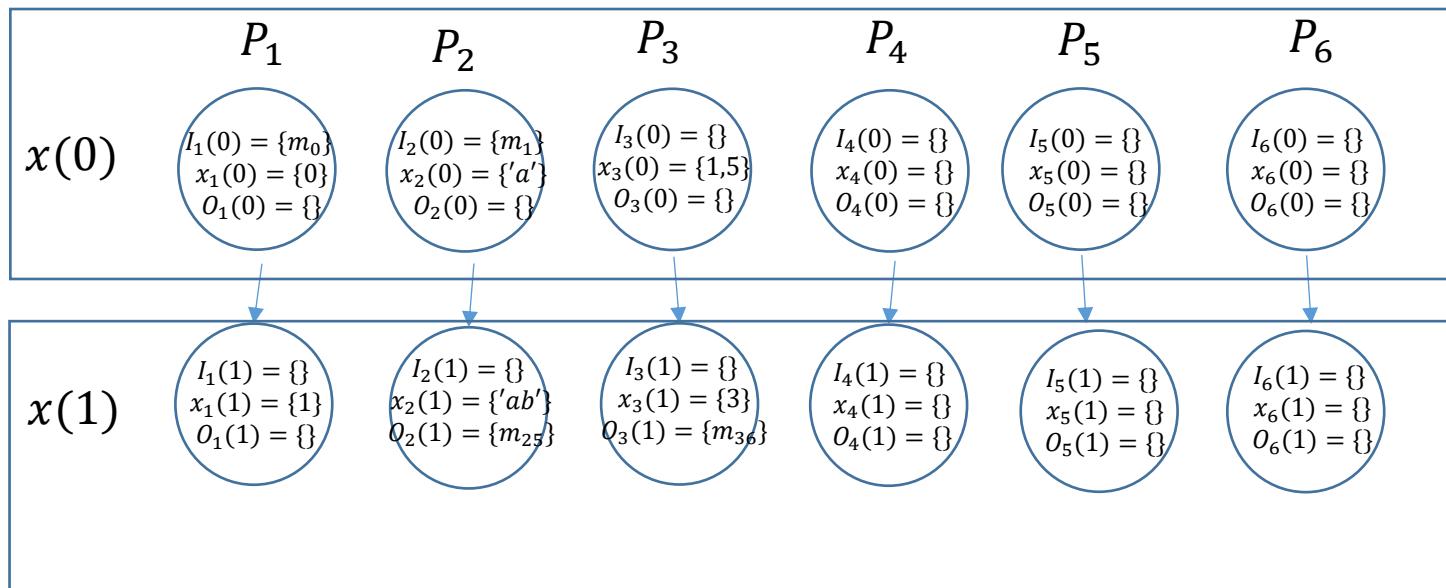
Modelul principal în SD sincron



Evenimente posibile:

- Operații locale pe baza $I_i(t)$ și $x_i(t)$: e.g. calcule numerice
- Evenimente de livrare de mesaje

Modelul principal în SD sincron



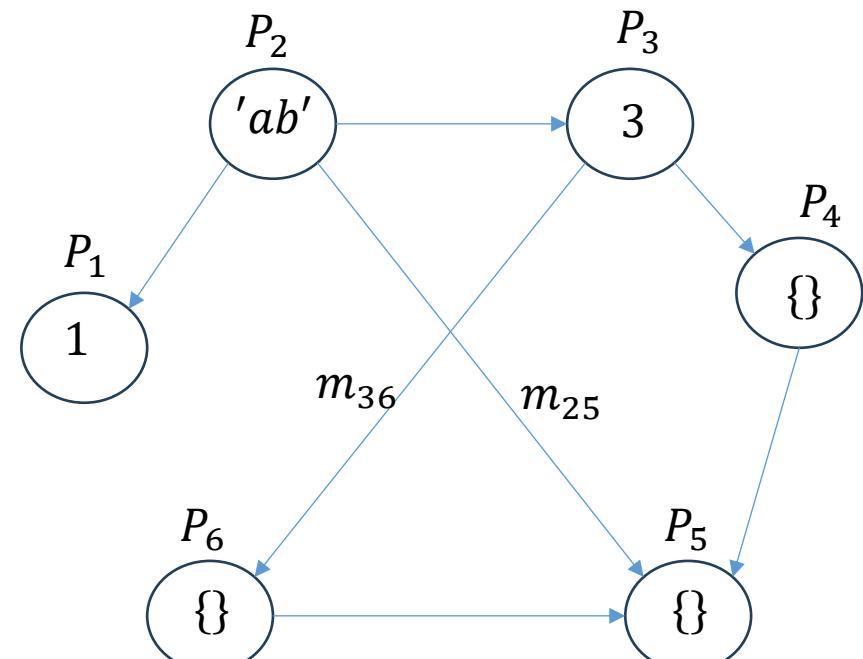
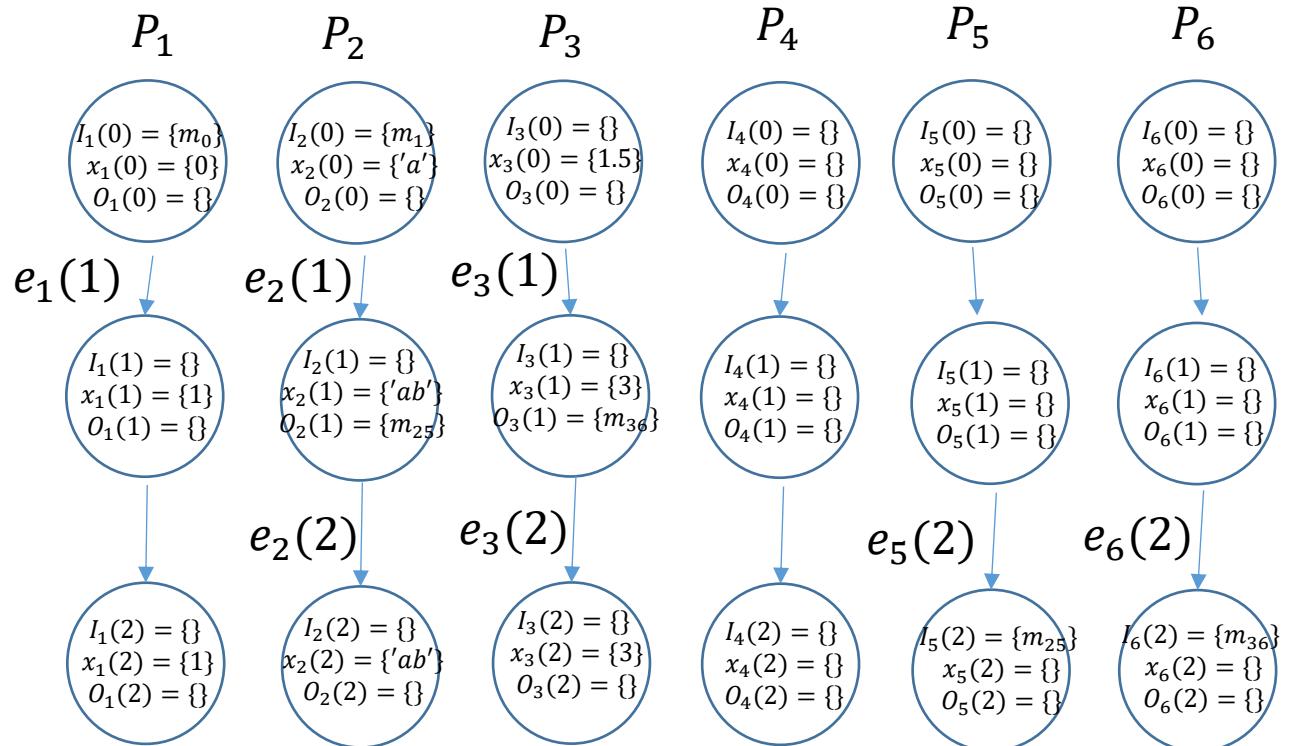
Starea globală a sistemului la momentul t :

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \\ x_5(t) \\ x_6(t) \end{bmatrix}$$

Starea inițială a sistemului:

$$x(0) = \begin{bmatrix} 0 \\ a \\ 1,5 \\ \text{NULL} \\ \text{NULL} \\ \text{NULL} \end{bmatrix} \Rightarrow \quad x(1) = \begin{bmatrix} 1 \\ ab \\ 3 \\ \text{NULL} \\ \text{NULL} \\ \text{NULL} \end{bmatrix}$$

Modelul principal in SD sincron



Modelul principal în SD sincron

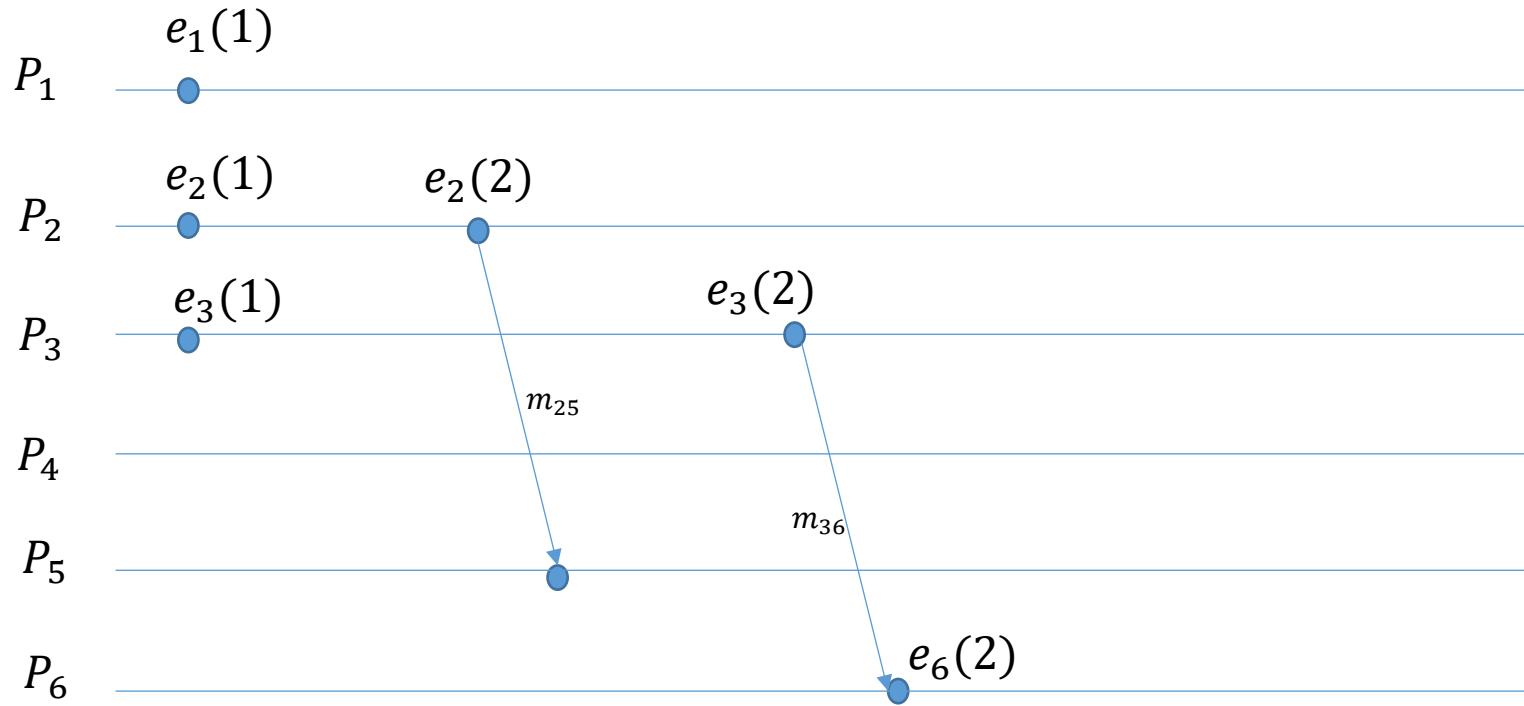


Diagrama spatiu-timp:

- Fiecare proces are propria vedere locală asupra evenimentelor
- Exprimă precedența cauzală între evenimente

Executie - traекторie

Rețele 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 traекторie/execuție: un sir (in)finit

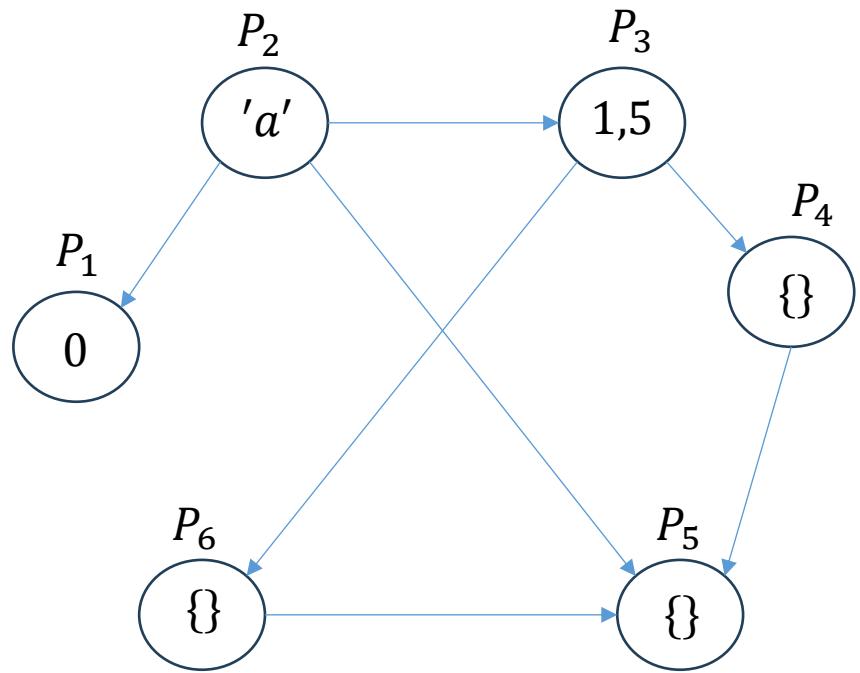
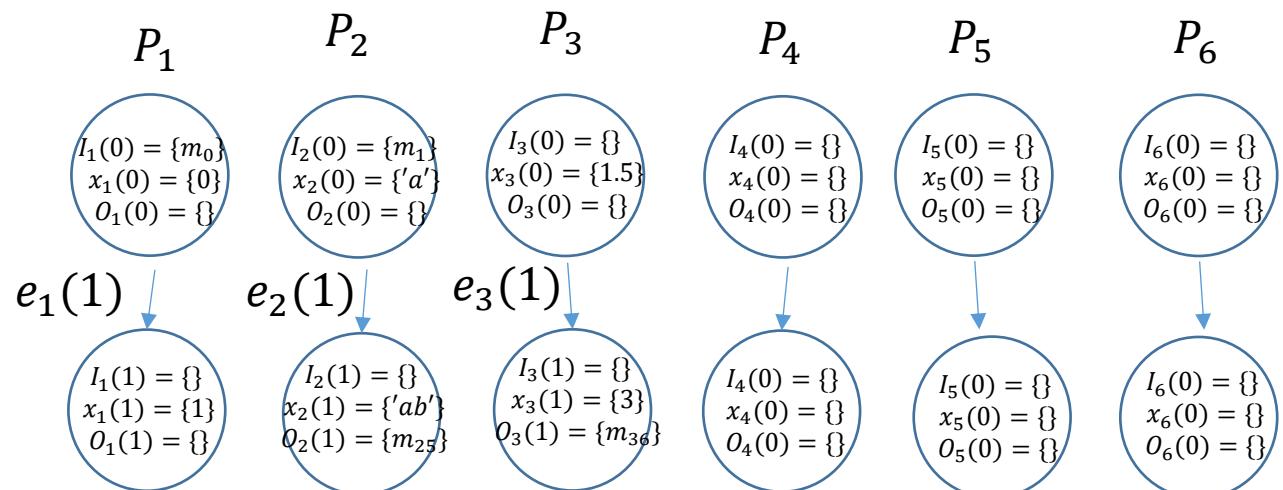
$$x(0), e(1), x(1), e(2), x(2), \dots \dots$$

Rețele sigure: nu se iau în calcul pierderi de pachete sau defecte

- O traекторie/execuție: un sir (in)finit

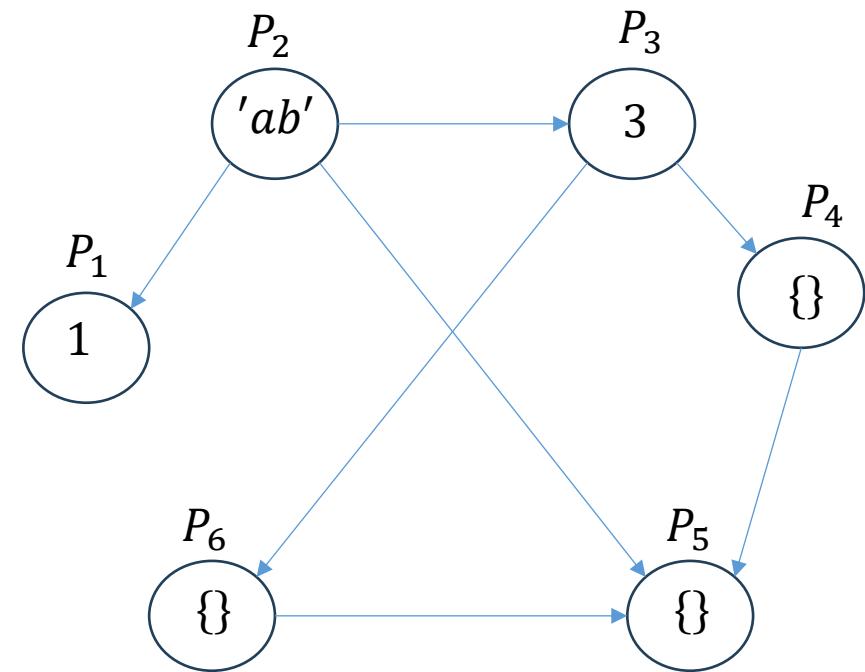
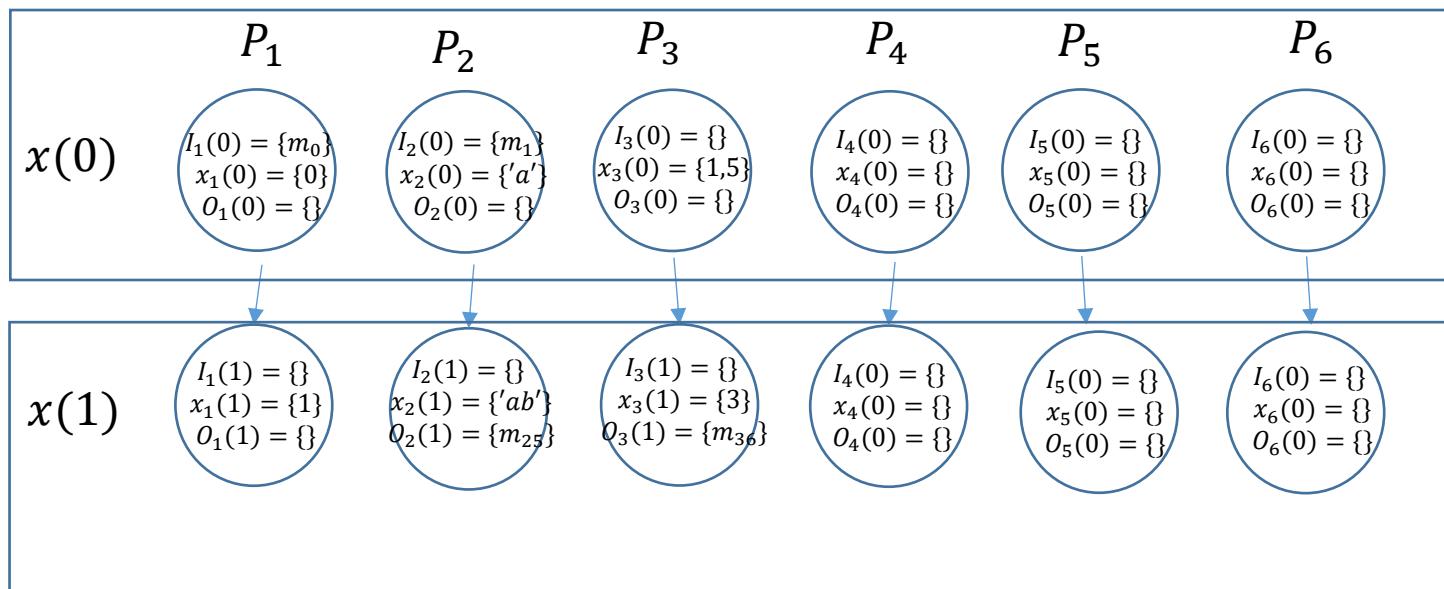
$$x(0), x(1), x(2), \dots \dots$$

Model in SD asincron



In contextual asincron, P_i are propriul ceas local, cu increment independent față de celelalte noduri.

Modelul principal în SD sincron



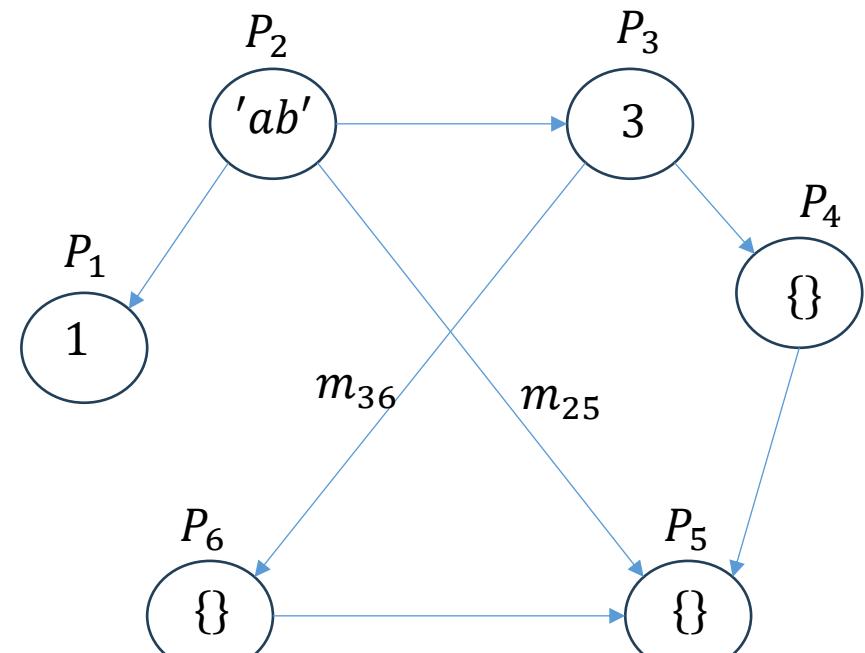
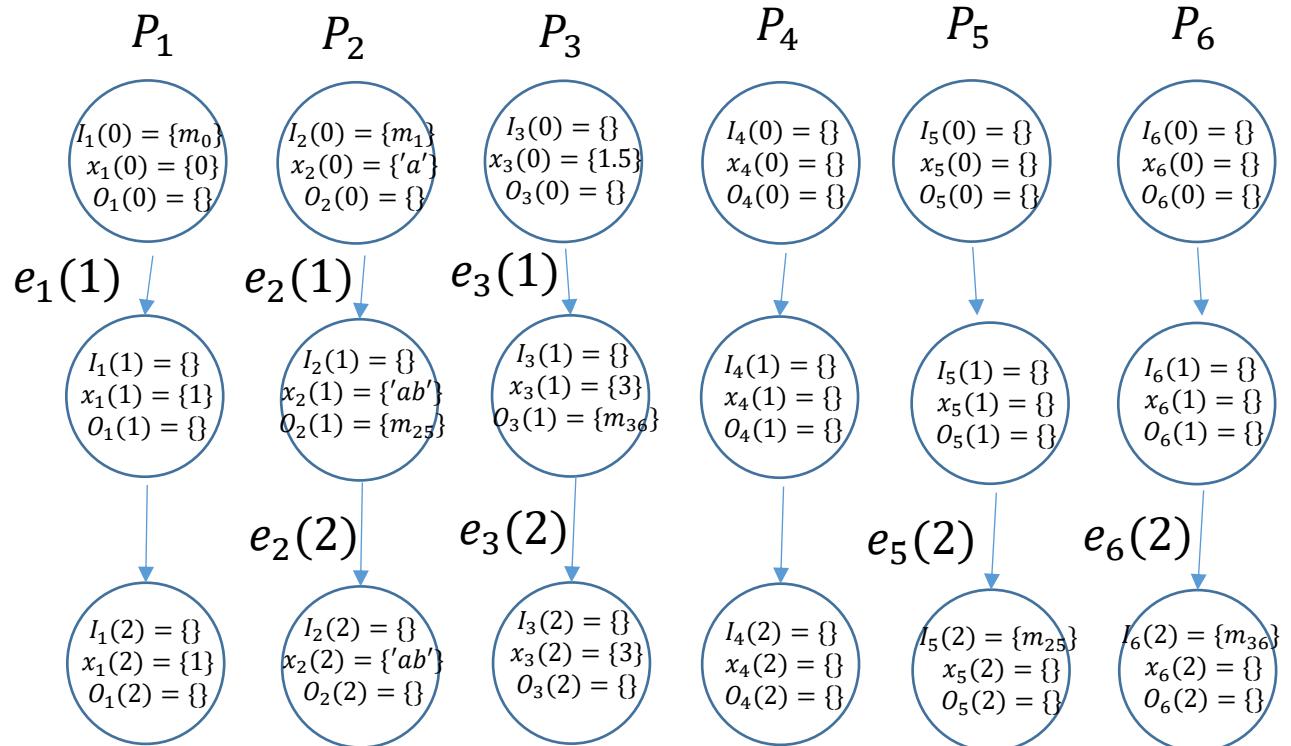
Starea globală a sistemului la momentul t :

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \\ x_5(t) \\ x_6(t) \end{bmatrix}$$

Starea inițială a sistemului:

$$x(0) = \begin{bmatrix} 0 \\ a \\ 1,5 \\ \text{NULL} \\ \text{NULL} \\ \text{NULL} \end{bmatrix} \Rightarrow x(1) = \begin{bmatrix} 1 \\ ab \\ 3 \\ \text{NULL} \\ \text{NULL} \\ \text{NULL} \end{bmatrix}$$

Modelul principal în SD sincron

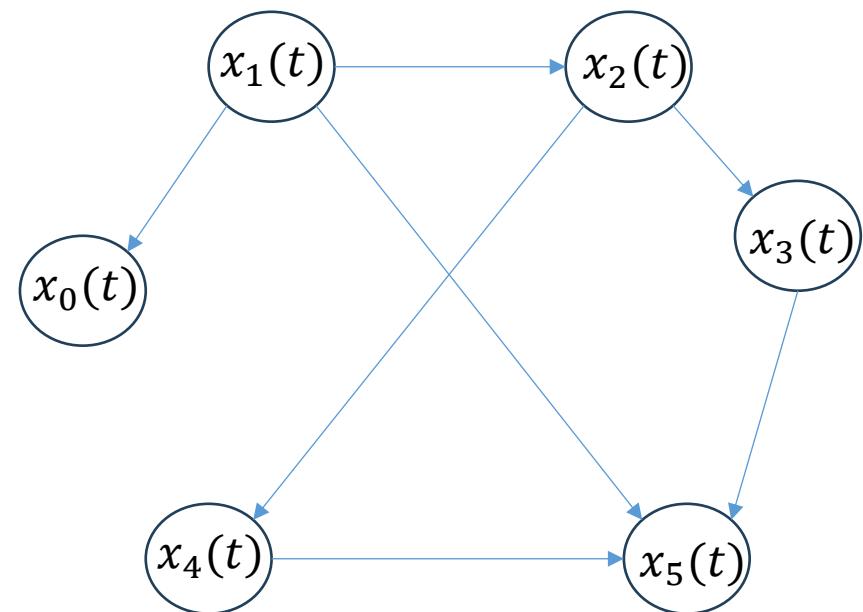


Sisteme și algoritmi distribuiți

Curs 2

Cuprins

- Modele de comunicație
- Ceasuri



Comunicație

În limbajul natural (uman) exprimarea unui mesaj cuprinde:

- Dialect
- Limbă
- Cuvînt
- Alfabet

Comunicație

Premise necesare pentru comunicație:

- Sursa și destinația sunt conectate fizic
- Sursa alege alfabetul, limba etc., în care codifică mesajul
- Destinația decodifică mesajul (recunoaște alegerile sursei)

Modele de comunicație

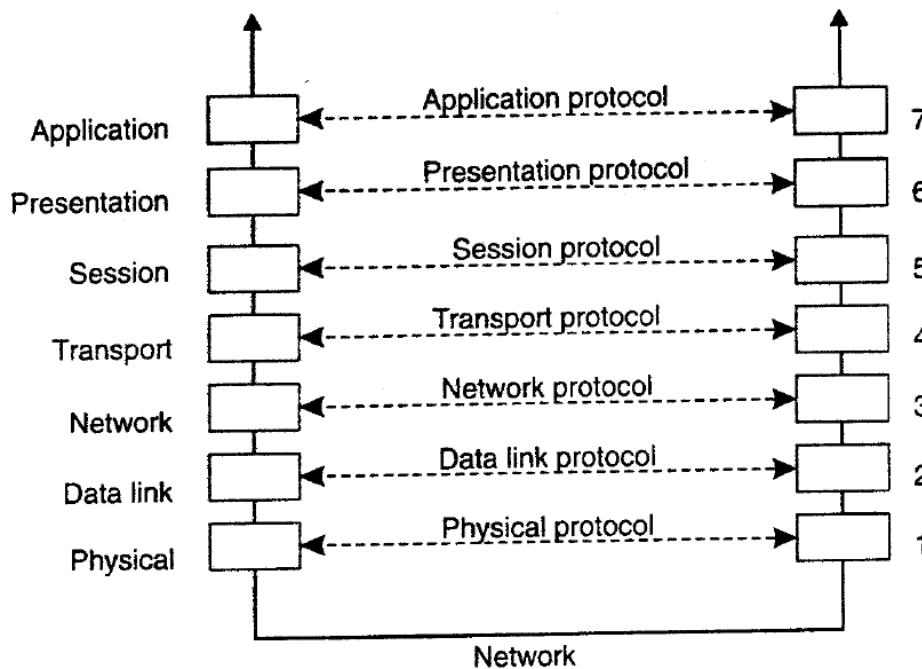
A comunică cu B:

- A produce un mesaj
- Apelează o primitivă pentru a trimite mesajul
- B apelează o primitivă pentru a primi mesajul



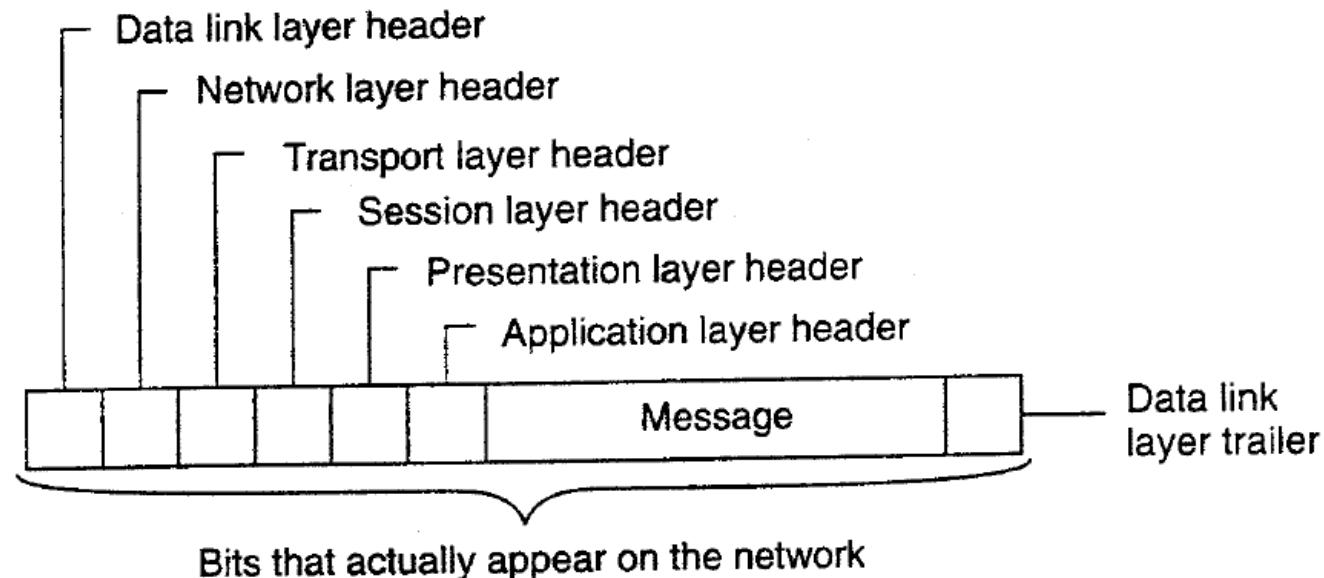
Protocole de comunicație

Protocol = algoritm care specifică formatul, conținutul și sensul mesajelor trimise și primite de către un proces.



Protocole de comunicație

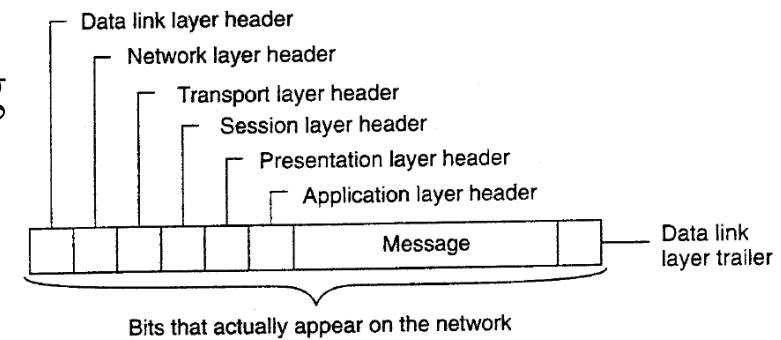
Protocol = algoritm care specifică formatul, conținutul și sensul mesajelor trimise și primite de către un proces.



Ex.: User Datagram Protocol (UDP)

Protocol de comunicație la nivelul (layer) de transport

- Fără conexiune = sursa comunică pachete către destinație fără a negocia o conexiune (implicit fără gestiunea pachetelor)
- Lightweight = nu necesită ordonare sau istoric de pachete; comunicația mesajului începe de la primul pachet
- Singurul mecanism de siguranță este *checksum*, nu există verificarea succesului transmisiei.
- Folosit în aplicațiile de streaming, broadcasting

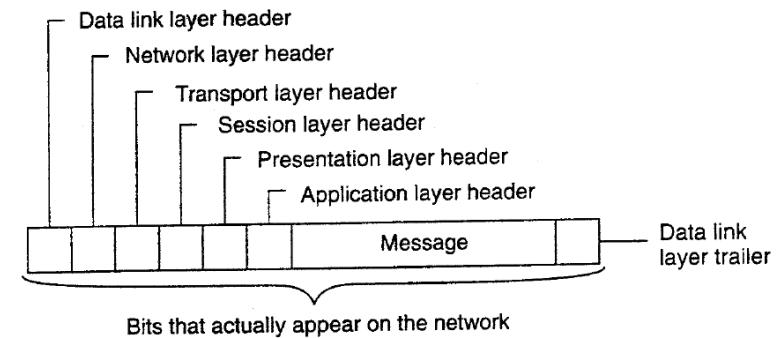


Ex.: User Datagram Protocol (UDP)

Ex.: Transmission Control Protocol (TCP)

Protocol de comunicație la nivelul (layer) de transport

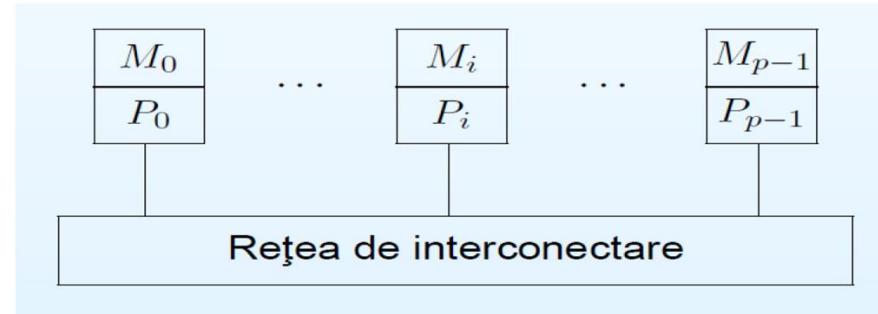
- Orientat pe conexiune = înainte de interschimbarea de mesaje dintre sursă și destinație, se realizează o conexiune.
- Heavyweight = primele 3 pachete necesare pentru conexiune prin socket.
- Mecanisme de ACK, retransmisie și time-out
- Cuprinde trei etape:
 - Stabilire conexiune
 - Transfer date
 - Oprită conexiune
- Divide mesajul în porțiuni și creează segmente de dimensiuni fixe



Ex.: Transmission Control Protocol (TCP)

Comunicația prin mesaje: Modelul SPMD

- MIMD cu memorie distribuia
- Paradigma **SPMD** (Single Program Multiple Data): toate procesoarele execută același program, dar fiecare utilizează un set propriu de date.
- În general, execuția programului nu este sincronă;
- Deși toate procesoarele văd același program, instrucțiunile executate nu sunt identice



Modelul SPMD

- Procesoarele sunt numerotate $0, \dots, p$
- Numerotarea nu este statică, ci se realizează la momentul execuției.
- Există primitive care returnează adresa unui procesor (e.g. MPI_rank)
- Procesoarele pot executa instrucțiuni diferite în funcție de adresa lor
 1. $\text{rank} \leftarrow$ adresa proprie
 2. **dacă** $\text{rank} = 0$ **atunci**
 1. $a \leftarrow 1$
 3. **altfel**
 1. $a \leftarrow 0$

Modelul SPMD - variabile

- O variabilă a unui program SPMD este multiplicată (de p ori) : fiecare procesor deține un exemplar, asupra căruia are control complet.
- Un procesor nu poate modifica variabile din memoria altui procesor.
- Putem interpreta variabila a ca un vector cu p elemente: fiecare procesor P_i deține componenta i a vectorului. Cu toate acestea i reprezintă un index global al datelor din a .
- Programul initializează a cu 0 pe toate componente, cu excepția primei componente (care este 1).

Modelul SPMD – problemă 1

Inițializați un vector a de dimensiune n cu zero, mai puțin elementul a_m , care să fie 1.

- Cum distribuim vectorul a celor p procesoare?
- Cum facem efectiv inițializarea?

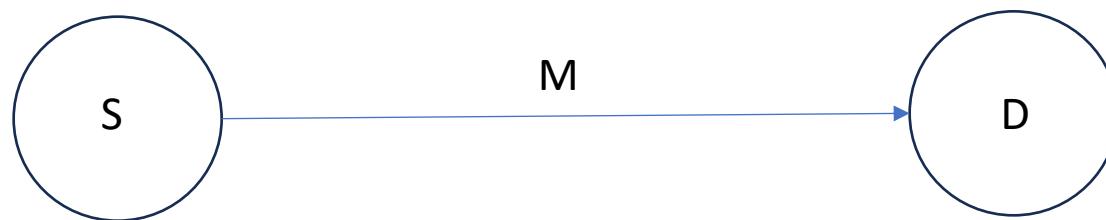
Modelul SPMD – problemă 2

Procesorul cu id 0 deține doi vectori (reali) u, v de dimensiune n . Realizați produsul scalar în mod distribuit.

- Cum distribuim vectorii u, v celor p procesoare?
- Ce operații de calcul intern realizează fiecare procesor?
- Cum acumulăm rezultatul?

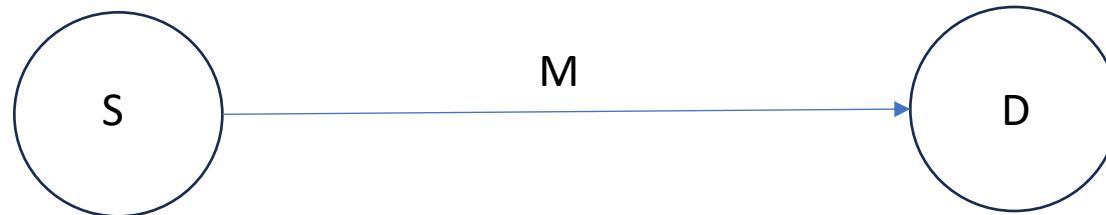
Modelul de comunicație prin mesaje

- Singura modalitate de comunicare între procesoare este transmiterea de mesaje
- **Operația de bază:**



Modelul de comunicație prin mesaje

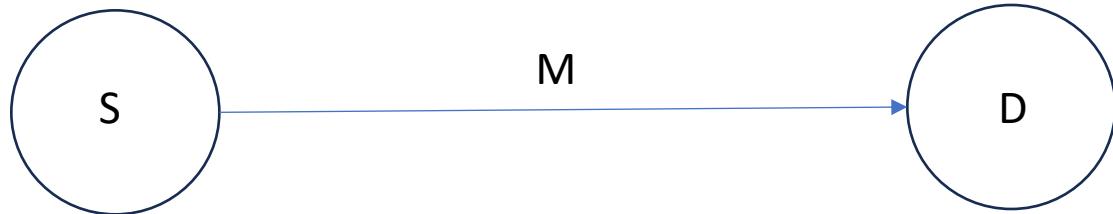
- **Operația de bază:**



- Procesorul sursă transmite prin rutina **send**
- Procesorul destinație recepționează prin rutina **recv**
- Sintaxă generală:
 - `send(date, dest)`
 - `recv(date, sursa)`
- date = locație (buffer) din care se preiau/depun mesajele transmise
- sursa/dest = adresa procesorului cu care se comunică

MP - corectitudine

- **Operația de bază:**



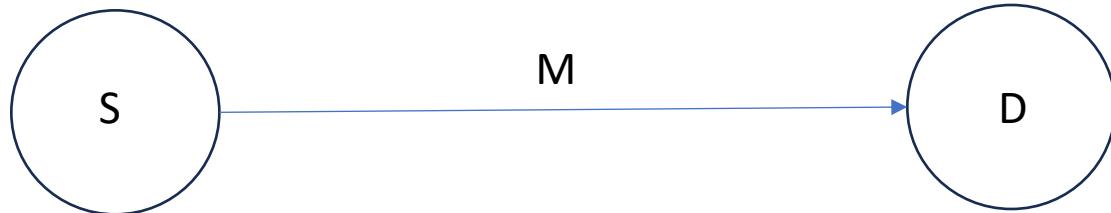
- Orice operație de **send** trebuie însoțită de una de **recv**
- Pe ansamblu, vom avea perechi **send-recv**

Exemplu: Procesorul i transmite un mesaj M vecinilor de la stânga, respectiv dreapta (pe o topologie inel)

1. **dacă** $\text{rank} = i$ **atunci**
 1. $\text{send}(M, (i + 1)\text{mod } p);$
 2. $\text{send}(M, (i - 1)\text{mod } p);$
2. **Altfel dacă** $\text{rank} = (i + 1)\text{mod } p$ **atunci** $\text{recv}(M, i);$
3. **Altfel dacă** $\text{rank} = (i - 1)\text{mod } p$ **atunci** $\text{recv}(M, i);$

MP - sincronizare

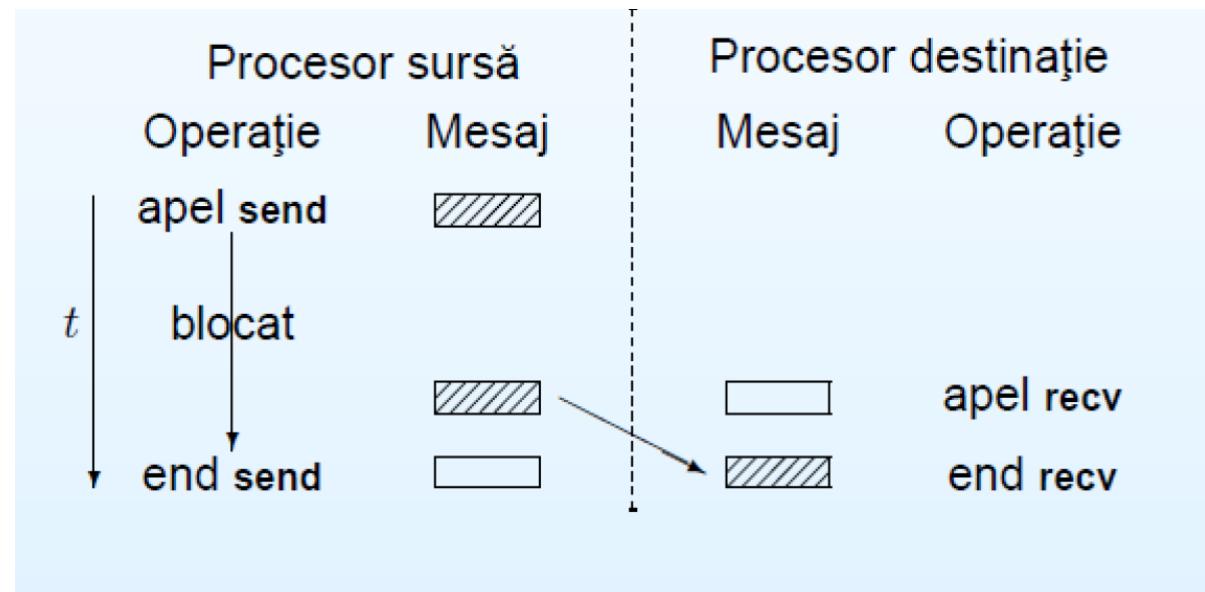
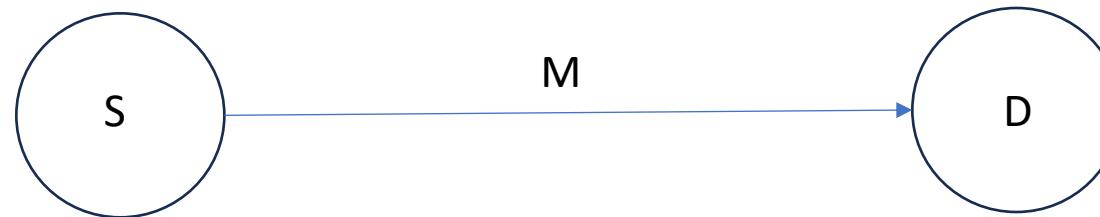
- **Operația de bază:**



- Momentul apelului primitivei **send** este în general diferit față de momentului apelului **recv**
- Ce se întâmplă din momentul apelului primei primitive până la finalizarea comunicației?
 - Răspunsuri posibile: (i) așteaptă (**comunicație blocantă**); (ii) poate executa alte operații (**comunicație non-blocantă**)

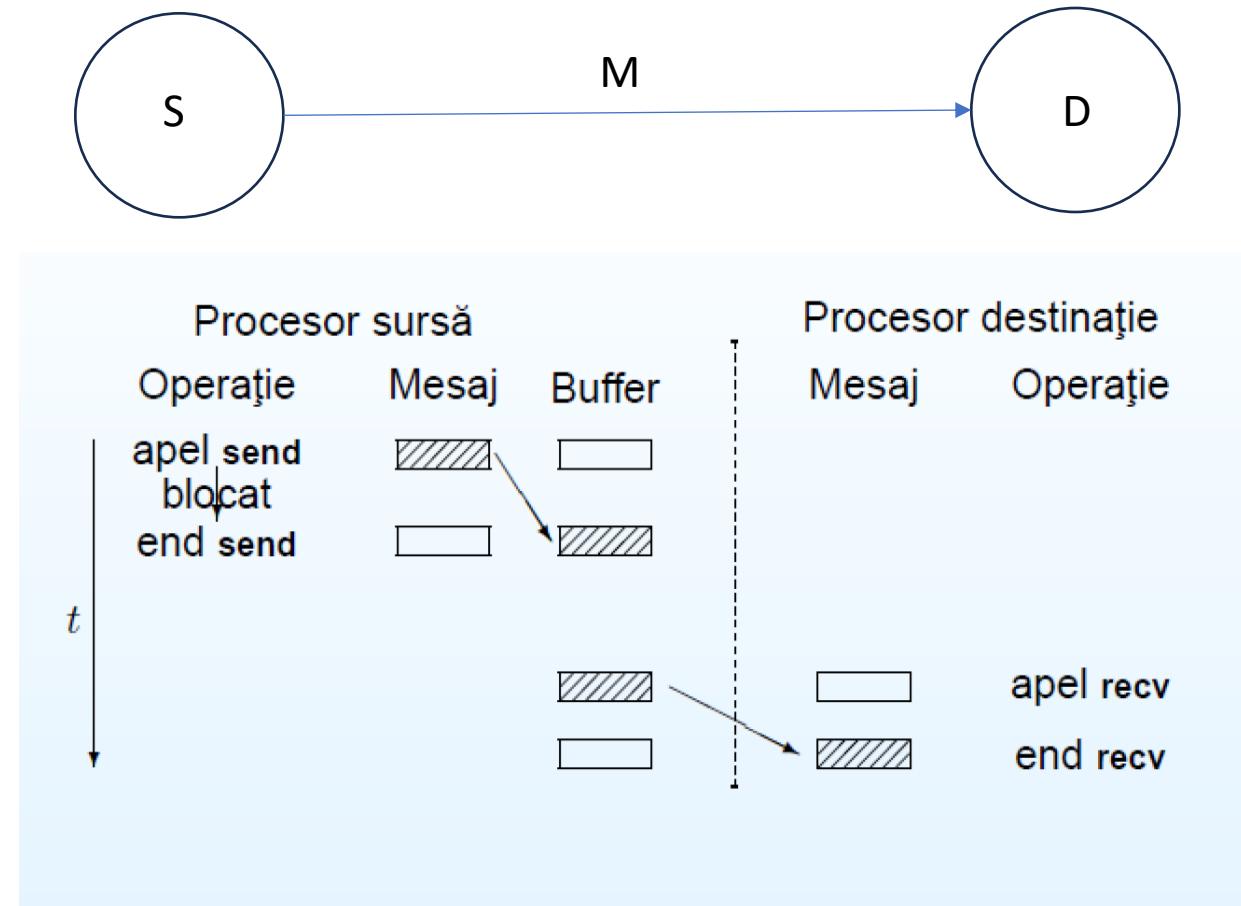
MP – comunicatie blocantă

- Operația de bază:



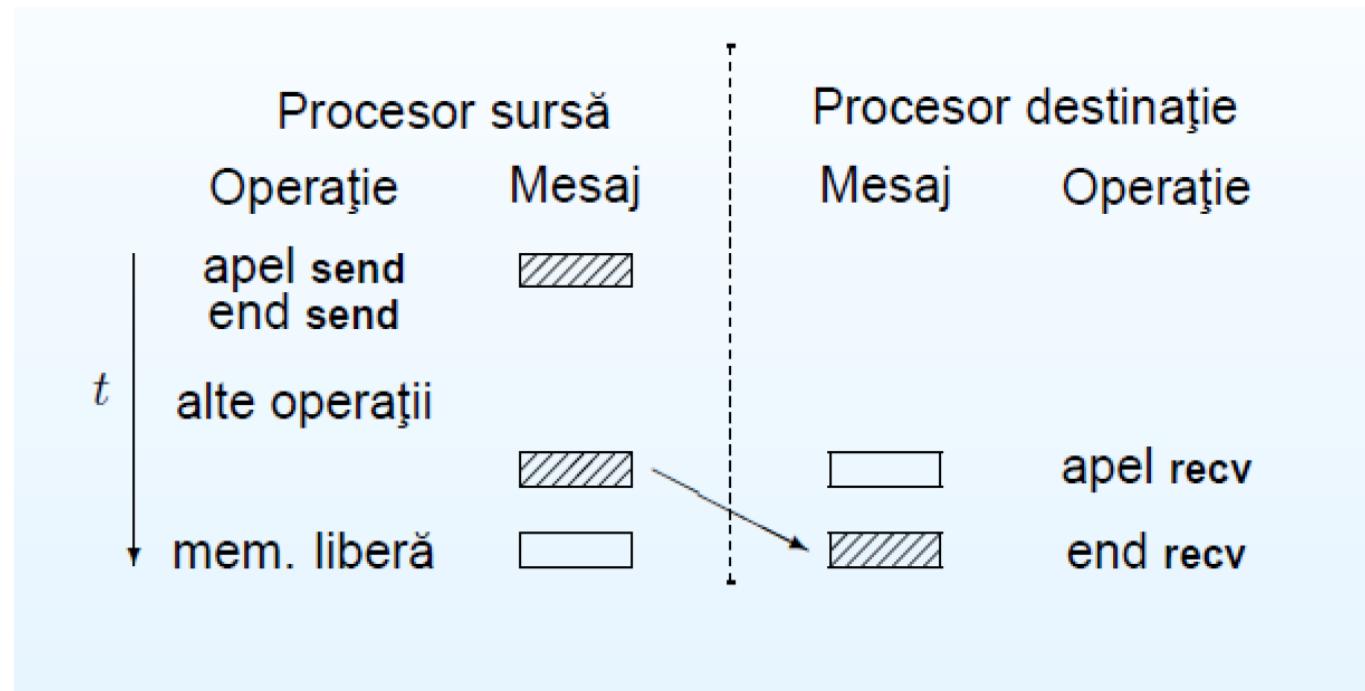
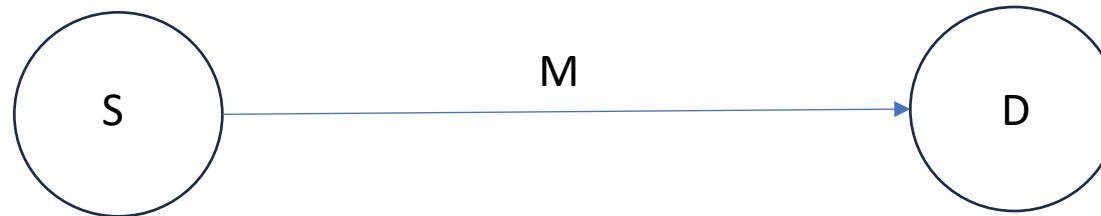
MP – comunicatie blocantă prin buffer

- Operația de bază:



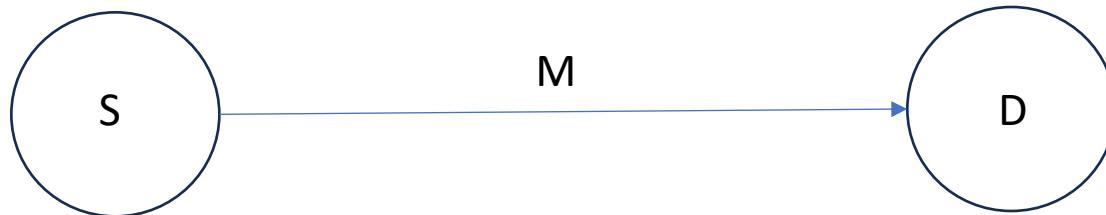
MP – comunicație non-blocantă

- Operația de bază:



MP – comunicație non-blocantă

- **Operația de bază:**



Alte primitive:

- **Așteaptă (Wait)** – așteptarea terminării comunicăției

Utilizare:

1. **send(*date*, dest);**
2. execută operații care nu modifică date;
3. **așteaptă terminarea send;**
4. modifică date;

MP – comparație

- Comunicația non-blocantă asigură o ocupare mai bună a procesoarelor
- Erorile de programare:
 - Comm blocantă: blocarea execuției
 - `send(m1, (rank + 1) mod p)`
 - `recv(m2, (rank - 1) mod p)`
 - Comm non-blocantă: alterarea zonei de memorie după apelul **send**

Standardul MPI (Message-Passing Interface)

- Standard care descrie primitivele de comunicație în paradigma SPMD
- Implementări: OpenMPI, mpich2 etc.
- O rutină MPI se execută în cadrul unui grup de procesoare (comunicator)
- Într-un comunicator procesoarele sunt numerotate $\{0, \dots, p - 1\}$

C:

```
int MPI_Comm_rank(MPI_Comm com, int * my_rank)  
int MPI_Comm_size(MPI_Comm com, int *p)
```

Python:

```
int comm.Get_rank()  
int comm.Get_size()
```

Exemplu MPI

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

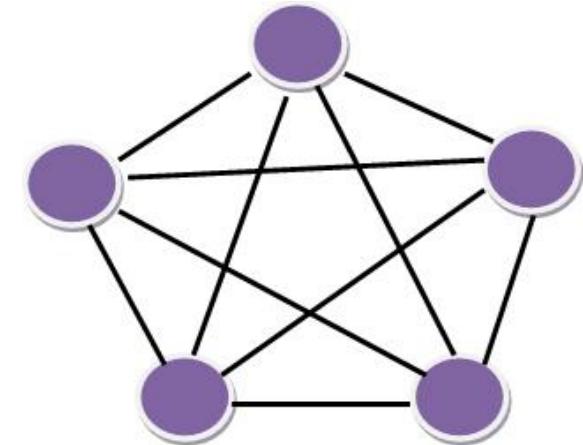
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

MPI primitives

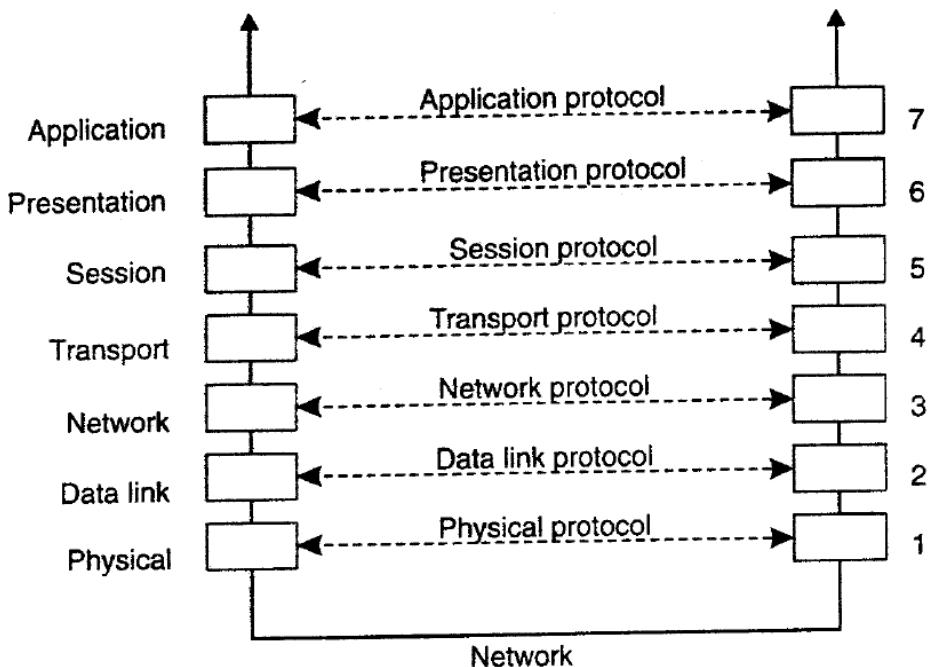
Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Modele de comunicație

1. Comunicație prin mesaje
2. *Comunicație prin mesaje persistente*



http://4.bp.blogspot.com/-LlvGwlSz_-4/UMadzwP7tI/AAAAAAAAY2AxHYokPHzk/s1600/all+channel.jpg



Comunicație prin mesaje persistente (MQS)

Message Queuing Systems = sistem în care aplicațiile comunică prin inserarea de mesaje în cozi specifice;
“O abstractizare a căsuței poștale”

- Sursa (producer) are garanția că mesajul său va fi eventual inserat în coada destinatarului (receiver)
- Nu avem garanții despre momentul când (sau dacă) mesajul va fi citit
- Sursa și destinatarul execută complet independent unul de celălalt.

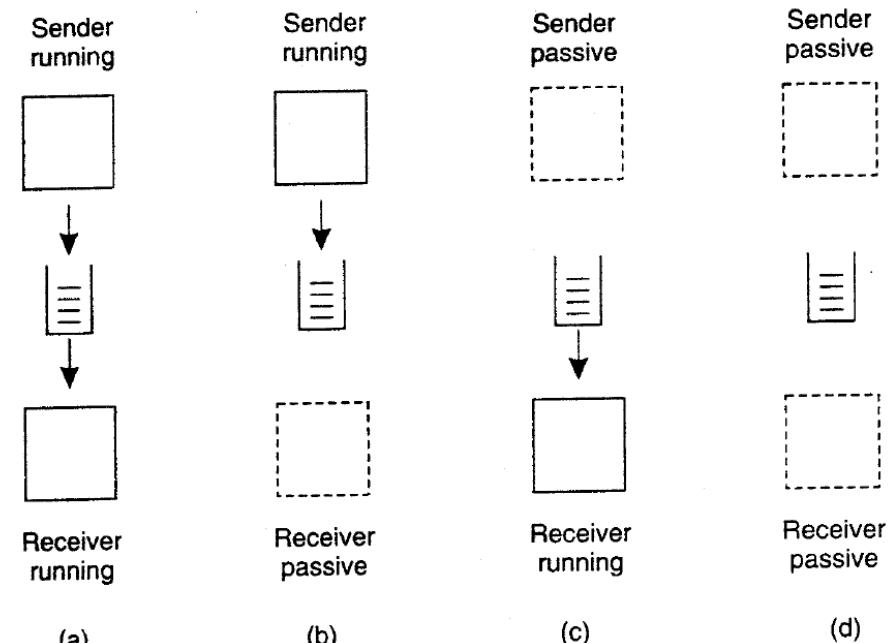


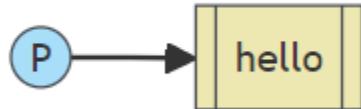
Figure 4-17. Four combinations for loosely-coupled communications using queues.

MQS primitives

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Figure 4-18. Basic interface to a queue in a message-queuing system.

Rabbit MQ example - sender



```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
```

- Client RabbitMQ ⇒ modul Python pika
- Asigurăm existența cozii în care scriem mesajele (cu numele ‘hello’)

```
channel.queue_declare(queue='hello')
```

```
channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')
print(" [x] Sent 'Hello World!'")
```

```
connection.close()
```

Rabbit MQ example - receiver



```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
```

- Se realizează conexiunea identic
- Dacă nu cunoaștem a priori existența cozii create de sursă, asigurăm existența cozii în același fel

```
channel.queue_declare(queue='hello')
```

```
def callback(ch, method, properties, body):
    print(f" [x] Received {body}")
```

```
channel.basic_consume(queue='hello',
                      auto_ack=True,
                      on_message_callback=callback)
```

```
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

```
if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

Rabbit MQ example

Execuție pe receiver:

```
python receive.py  
# => [*] Waiting for messages. To exit press CTRL+C
```

Execuție pe sender:

```
python send.py  
# => [x] Sent 'Hello World!'
```

```
# => [*] Waiting for messages. To exit press CTRL+C  
# => [x] Received 'Hello World!'
```

References

Seif Haridi, <https://canvas.instructure.com/courses/902299/modules>

A.S. Tanenbaum, M.V. Steen, *DISTRIBUTED SYSTEMS: Principles and Paradigms*, Pearson Prentice Hall, Second Edition, 2007.

A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, Cambridge University Press, 2008.

M. Raynal, *Distributed Algorithms for Message-Passing Systems*, Springer-Verlag, 2013.

Sisteme și algoritmi distribuiți

Curs 3

Cuprins

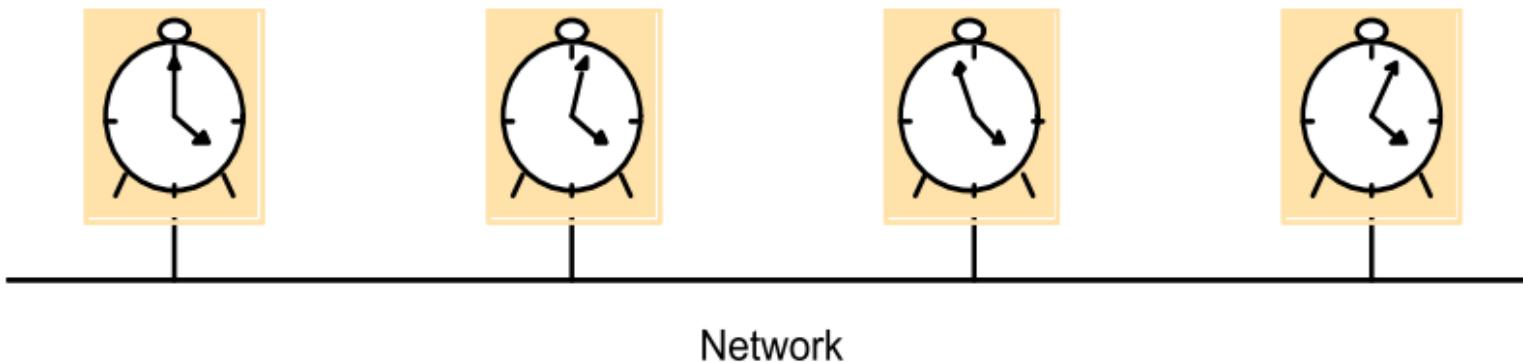
- **Ceasuri**
- Algoritmi sincroni
- Alegere lider pe inel (sincron)
- Alegere lider pe topologii generale (sincron)

Timp

- Ceasurile fizice în computere sunt realizate prin contorizare
 - Ceasuri atomice: drift 1s/150 milioane de ani
 - Ceasuri de sistem
 - Ceasuri de timp real: alimentate prin baterie (funcționează chiar dacă sistemul este oprit)
- $h(t)$ rata (viteza) ceasului hardware
- $H(t) = \int_0^t h(\tau)\tau$ valoarea ceasului hardware
- Ceas logic (registru): $C(t) = \alpha H(t) + \beta$
- $C(t)$ este un scalar crescător și se actualizează prin citiri ale lui $H(t)$

Timp

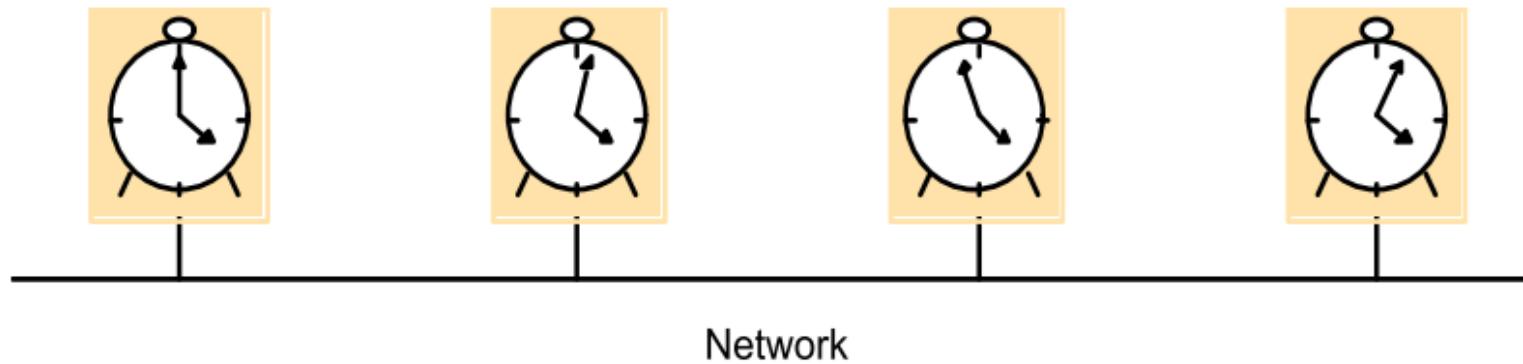
- În SD, ceasurile locale sunt **decalate** și **întârziante**
- **Decalaj** între nodurile (i, j) la momentul t: $|C_i(t) - C_j(t)|$
- **Întârziere** între nodurile (i, j) la momentul t: $|\frac{d}{dt} C_i(t) - \frac{d}{dt} C_j(t)|$



Problema sincronizării

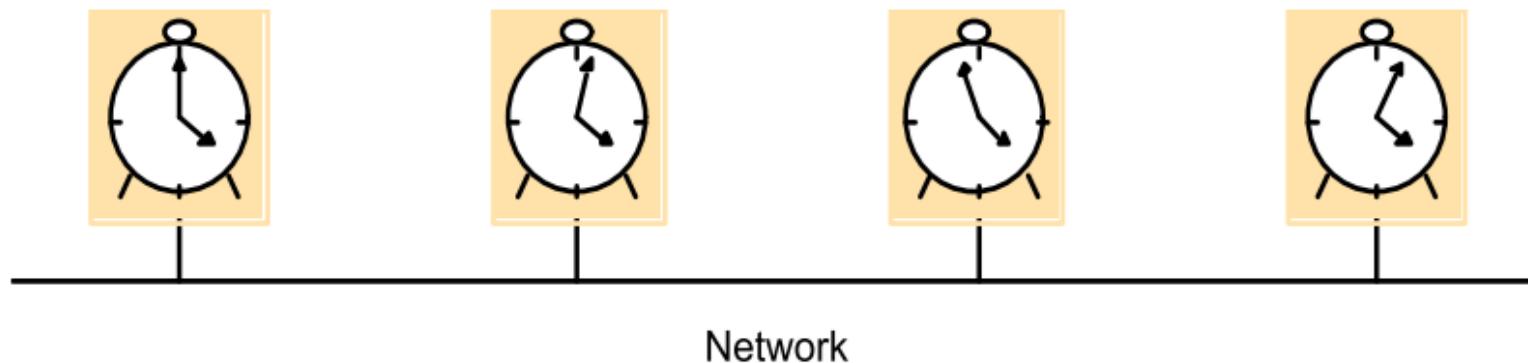
Folosind referințe externe sau interne, problema sincronizării ceasurilor se reduce la asigurarea relației (de precizie):

$$|C_i(t) - C_j(t)| \leq \rho \quad \forall t \geq t_0$$



Problema sincronizării

- Sincronizare externă: referința este un ceas absolut extern e.g. Coordinate Universal Time (UTC), GPS etc.
- Sincronizare internă: referința este o valoare stabilită în rețea
 - Algoritmul lui Cristian
 - Algoritmul de Medie (Berkeley Algorithm)



UTC

UTC este standardul primar de timp în lume, transmis prin: radio, linii telefonice line, satelit (GPS) etc.

Protocolele populare de sincronizare folosesc referința UTC și urmăresc să asigure relația (acuratețe):

$$|S(t) - C_i(t)| < \rho \quad \forall i, \forall t$$

Dacă asigurăm acuratețe ρ atunci ce precizie garantăm?

Algoritmul lui Cristian

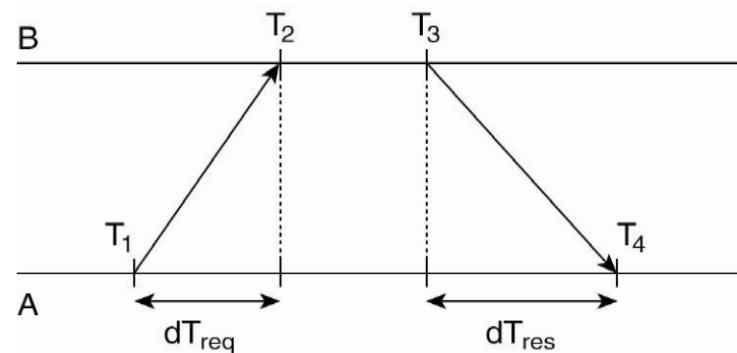
- Ipoteza 1: Întârzieri pe comunicație simetrice și mărginite (rețele LAN)
- Ipoteza 2: Există un nod de referință (pasiv) R cu unicul rol de a furniza referință
- Nodurile care nu sunt referință se vor sincroniza cu ceasul referinței

Algoritmul lui Cristian

Procedura AC: Nodul P cere periodic valoarea timpului de la R:

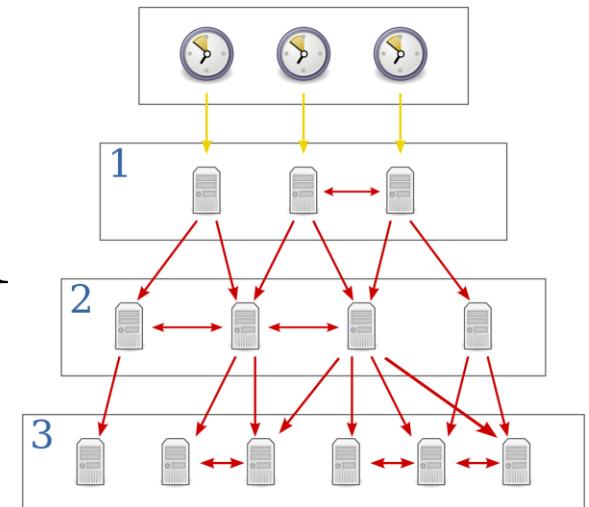
- T_1 : send request; T_4 : primește reply
- P primește val. T_2 și T_3 de la R, ajustează $C(t) = T_3 + T_{res}$ (T_{res} timp livrare mesaj)
- Folosește estimarea $T_{res} \approx \frac{T_{req}+T_{res}}{2}$

$$T_{res} = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$



Network Time Protocol (NTP)

- Implementare standard în rețele a alg. lui Cristian
- Peste multiple măsurători, calculăm T_{res} minim
- Ierarhie a serverelor de timp:
 - un TS de nivel k se sincronizează după unul de nivel $\leq k - 1$
 - Nivelul 0 este referința UTC

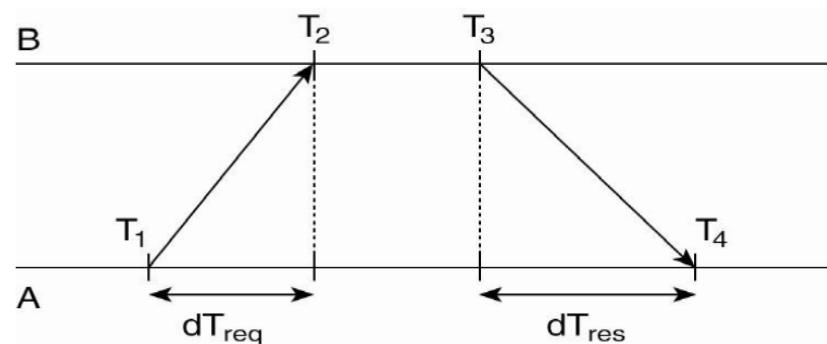


Exemplu

La 5:08:15.100, P trimite mesaj de request către R. La 5:08:15.900, P primește răspuns de la B cu valoarea 5:09:25.300 ($T_2 = T_3$)

- Care este valoarea ajustată a ceasului local $T_4 = C_P(t)$ după sincronizare?

$$T_{res} = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

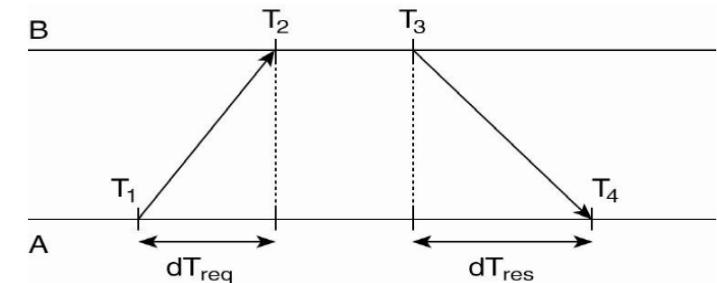


Exemplu

La 5: 08: 15.100, P trimite mesaj de request către R. La 5: 08: 15.900, P primește răspuns de la B cu valoarea 5: 09: 25.300 ($T_2 = T_3$)

- Send req la $T_1 = 5: 08: 15.100$
 - Primește răsp. la $T_4 = 5: 08: 15.900$
 - Mesajul este $T_3 = T_2 = C_R(t) = 5: 09: 25.300$
-
- Durata totală $T_4 - T_1 = 800 \text{ ms}$
 - Estimare: mesaj generat în urmă cu 400 ms
 - Set $C_P(t) = T_{serv} + 400 = 5: 09.25.700$

$$T_{res} = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

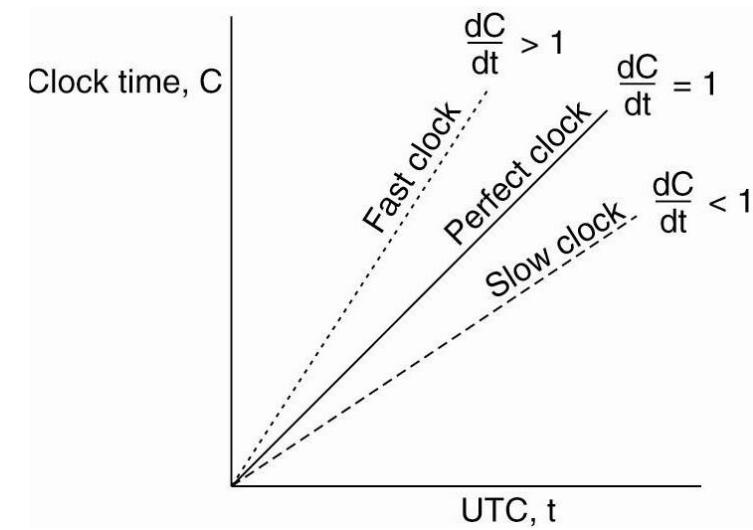


Sincronizare internă

Sincronizarea internă a ceasurilor locale în DS presupune (precizie)

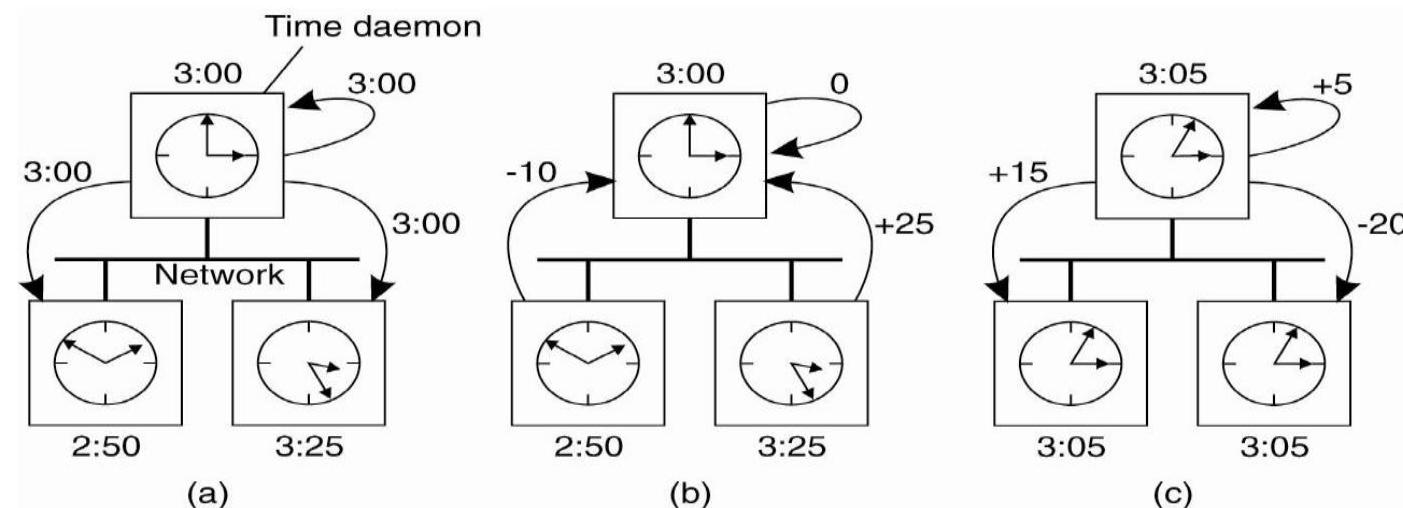
$$|C_j(t) - C_i(t)| < \rho \quad \forall i, j, t$$

- Necesită algoritmi complet distribuiți
- Problema este una de consens distribuit
- Vom reveni la ea în cursurile următoare



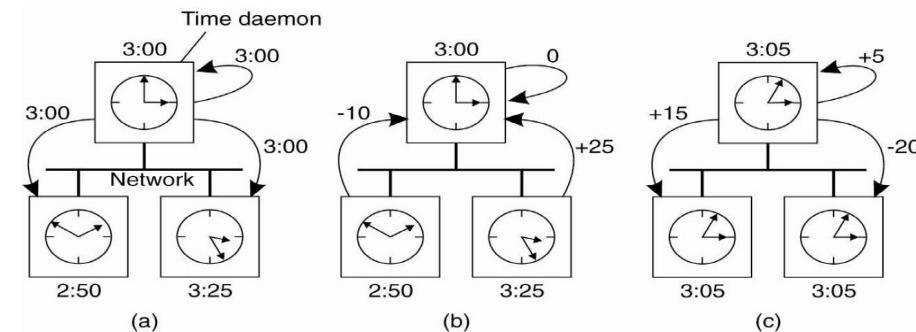
Algoritmul de medie (Berkeley Algorithm)

- Referință este unul din nodurile rețelei, ales eventual prin proceduri de leader-election
- Restul nodurilor urmăresc alinierea ceasurilor cu referința (consens)



Algoritmul de medie (Berkeley Algorithm)

- Referință este unul din nodurile rețelei, ales eventual prin proceduri de leader-election
- Restul nodurilor urmăresc alinierea ceasurilor cu referință (consens)
- Pe scurt: la iterația t
 - R difuzează valoarea $C_R(t)$
 - P_i calculează întârzierea locală $\delta_i = |C_R(t) - C_i(t)|$ și răspunde lui R
 - R distribuează ajustările pentru $C_i(t)$



Alternative

1. If two machines don't interact, there is no need to synchronize them (Leslie Lamport)
2. Dinamica întârzierii per link $\frac{T_{req} + T_{res}}{2}$ depinde de mai mulți factori
3. Adesea, contează ca procesele să convină asupra **ordinii** evenimentelor, și nu asupra **timpului** la care au avut loc (vezi Ceasuri Logice).

Cuprins

- Ceasuri
- **Algoritmi sincroni**
- Alegere lider pe inel (sincron)
- Alegere lider pe topologii generale (sincron)

Algoritmi sincroni în SD

Un algoritm distribuit sincron reprezintă un set de operații de calcul/comunicație *executat în iterații/runde* (contorizate de t) pe nodurile sistemului, cu scopul rezolvării unei sarcini concrete.

- Denumim starea locală a nodului i la momentul t cu $x_i(t)$. Scopul algoritmului este conducerea lui $x_i(t)$ către starea optimă $x_i(\infty)$
- La următorul moment de timp starea x_i suferă o transformare bazată pe pașii algoritmului și informația provenită de la vecini. Pe scurt,
$$x_i(t + 1) := f_i(x(t))$$
 unde $f_i(\cdot)$ reprezintă funcția de transformare asociată nodului i . Funcția $f_i(\cdot)$ definește însuși algoritmul SPMD.

Algoritmi sincroni în SD

La următorul moment de timp starea x_i suferă o transformare bazată pe pașii algoritmului și informația provenită de la vecini. Pe scurt,

$$x_i(t + 1) := f_i(x(t))$$

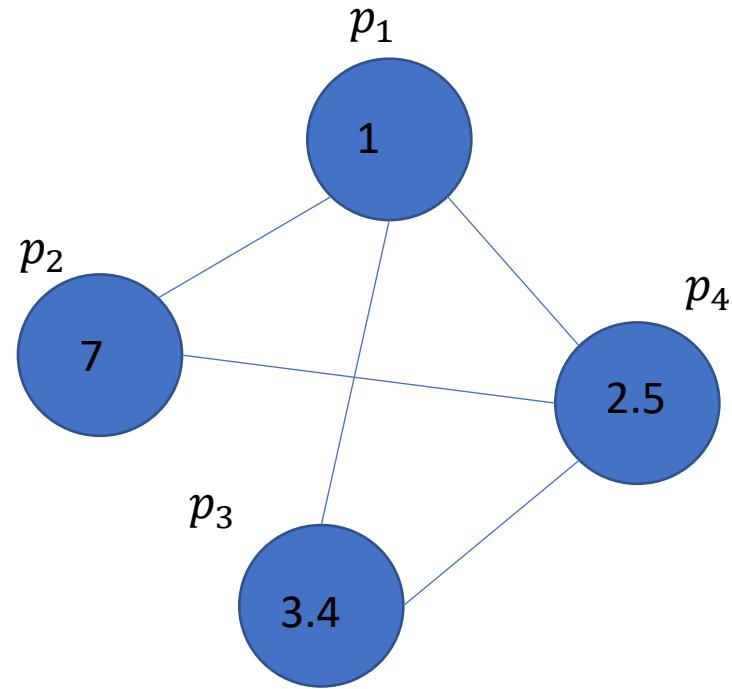
unde $f_i(\cdot)$ reprezintă funcția de transformare asociată nodului i .

Funcția $f_i(\cdot)$ este definită în format SPMD și este compusă din una sau mai multe operații (e.g. aritmetice, numerice) asupra variabilelor locale din memoria nodului i .

Medie sir de numere reale

Problemă [Distributed averaging]: Vectorul $x(0)$ este distribuit peste n noduri, astfel încât $x_i(0)$ se află în memoria locală a nodului i . Calculează distribuit media aritmetică $m = \sum_i x_i(0)/n$ a vectorului $x(0)$, încât la final $x_i(\infty) = m$.

$$x(0) = \begin{bmatrix} 1 \\ 7 \\ 3.4 \\ 2.5 \end{bmatrix}$$

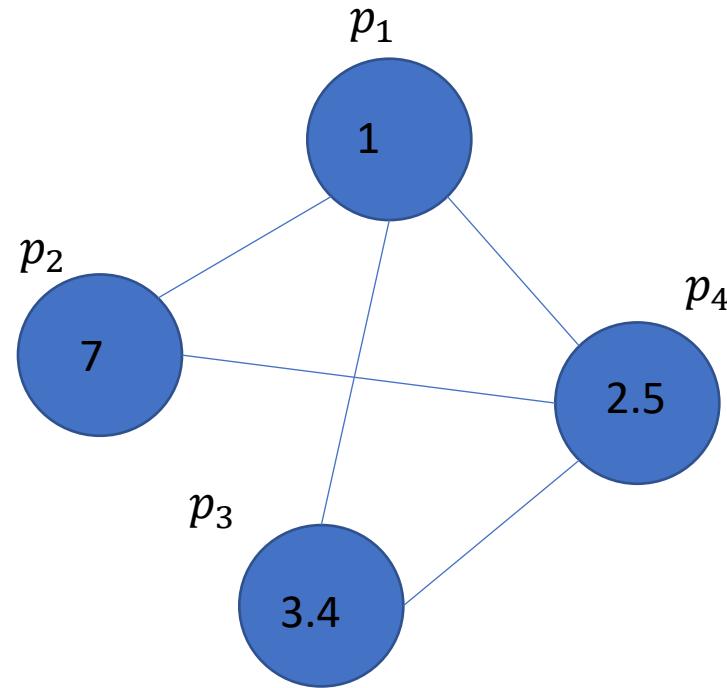


Medie sir de numere reale

Problemă [Distributed averaging]: Vectorul $x(0)$ este distribuit peste n noduri, astfel încât $x_i(0)$ se află în memoria locală a nodului i . Calculează distribuit media aritmetică $m = \sum_i x_i(0)/n$ a vectorului $x(0)$, încât la final $x_i(\infty) = m$.

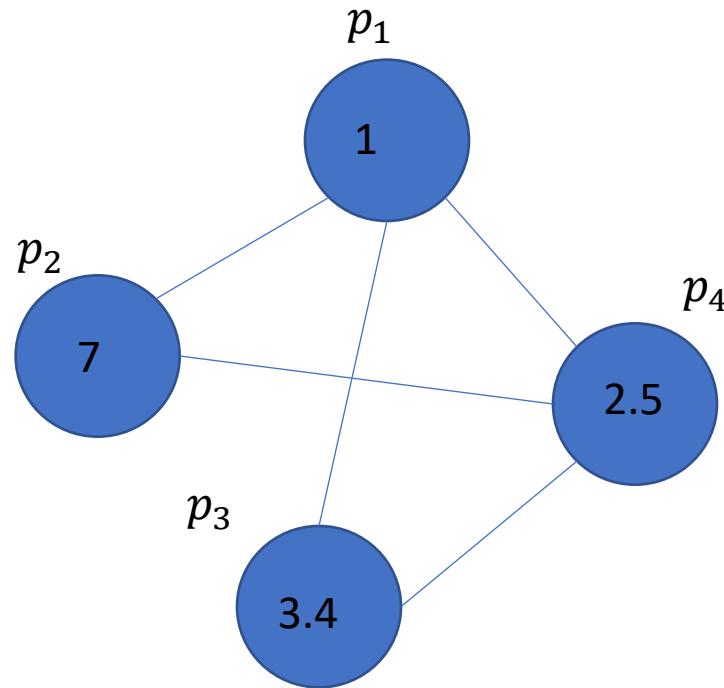
Algoritm: Alege $j \in N_i$ și actualizează $x_i(t + 1) = f_i(x(t)) = \frac{x_i(t) + x_j(t)}{2}$

$$x(0) = \begin{bmatrix} 1 \\ 7 \\ 3.4 \\ 2.5 \end{bmatrix}$$



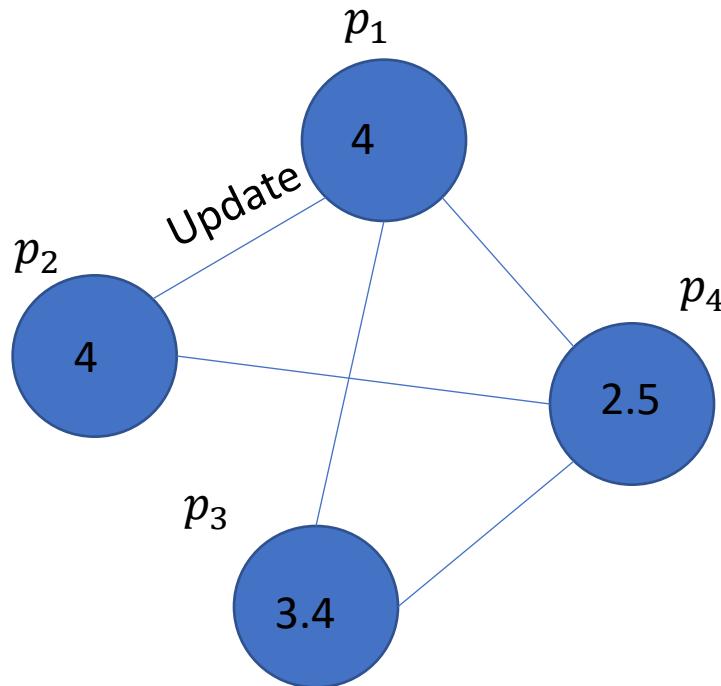
Medie sir de numere reale

$$x(0) = \begin{bmatrix} 1 \\ 7 \\ 3.4 \\ 2.5 \end{bmatrix}$$



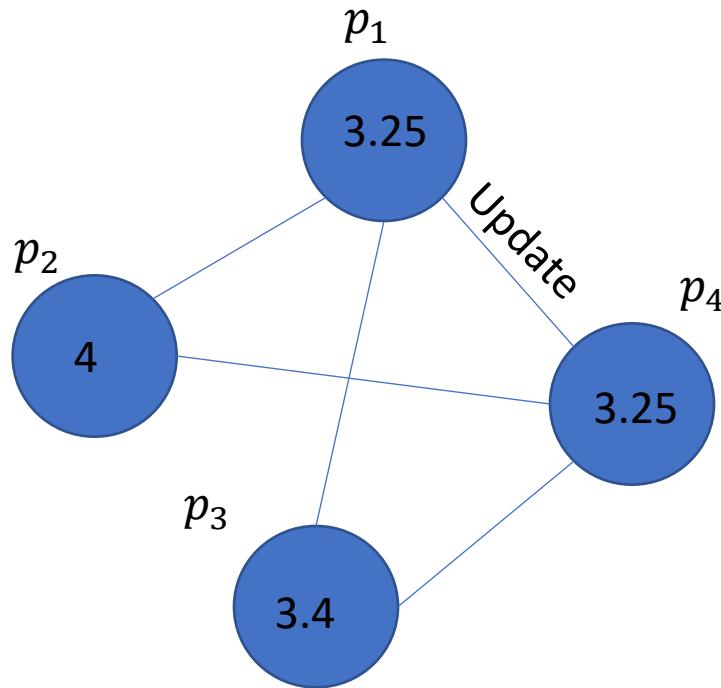
Medie sir de numere reale

$$x(1) = \begin{bmatrix} 4 \\ 4 \\ 3.4 \\ 2.5 \end{bmatrix}$$



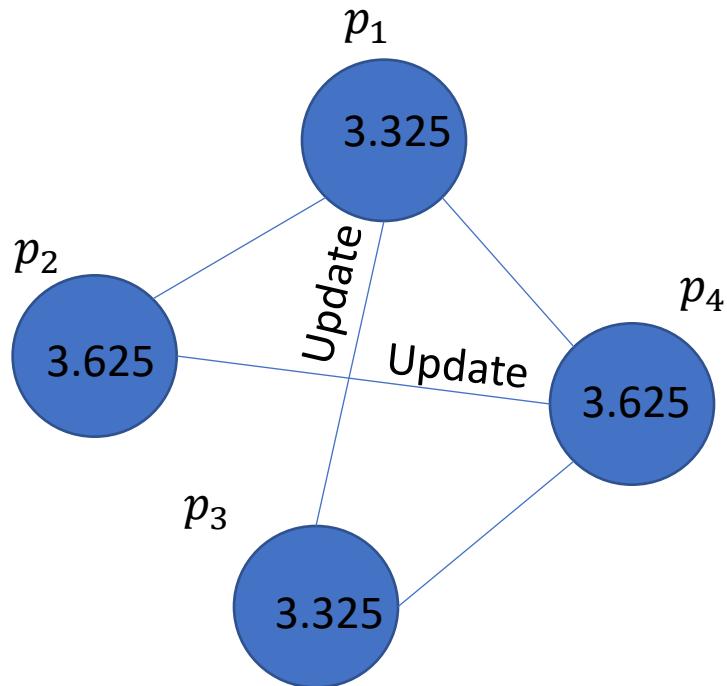
Medie sir de numere reale

$$x(2) = \begin{bmatrix} 3.25 \\ 4 \\ 3.4 \\ 3.25 \end{bmatrix}$$



Medie sir de numere reale

$$x(4) = \begin{bmatrix} 3.325 \\ 3.625 \\ 3.325 \\ 3.625 \end{bmatrix}$$



Algoritmi Sincroni în SD

La finalul iterației t fiecare nod finalizează calculul local, și comunicația cu vecinii, specifice iterației t

$$x(t+1) := \begin{bmatrix} x_1(t+1) \\ \dots \\ x_n(t+1) \end{bmatrix} = \begin{bmatrix} f_1(x(t)) \\ \dots \\ f_n(x(t)) \end{bmatrix} = F(x(t))$$

- Există o margine superioară pe timpul de comunicație între oricare două noduri.
- Adesea implementarea unui criteriu de oprire este o operație dificilă!

Alegere Lider (Leader Election)

În multe aplicații este necesară alegerea unui nod pentru operații particulare (e.g. difuzare, distribuție, master-slave).

Fiecare nod are un ID unic, ales dintr-un spațiu total ordonat.

Convenție: Lider = **nodul cu ID-ul maxim**.

Algoritmii de LE realizează *de facto* calculul distribuit al funcției
 $\max\{id_1, id_2, \dots, id_n\}$

Alegere Lider (Leader Election)

Starea locală a nodului i specifică calitatea de lider/non-lider:

$$x_i(t) \in \{\text{lider}, \text{non-lider}\}$$

Funcția de transformare asociată nodului i :

$$f_i(\{x_j(t) | j \in N_i\})$$

decide dacă la iterația curentă nodul i devine sau nu lider.

Funcția f_i se reduce la una sau mai multe operații asupra memoriei locale M_i a nodului P_i .

Alegere Lider (Leader Election)

Starea locală a nodului i specifică calitatea de lider/non-lider:
 $x_i(t) \in \{lider, non-lider\}$

Funcția de transformare asociată nodului i :

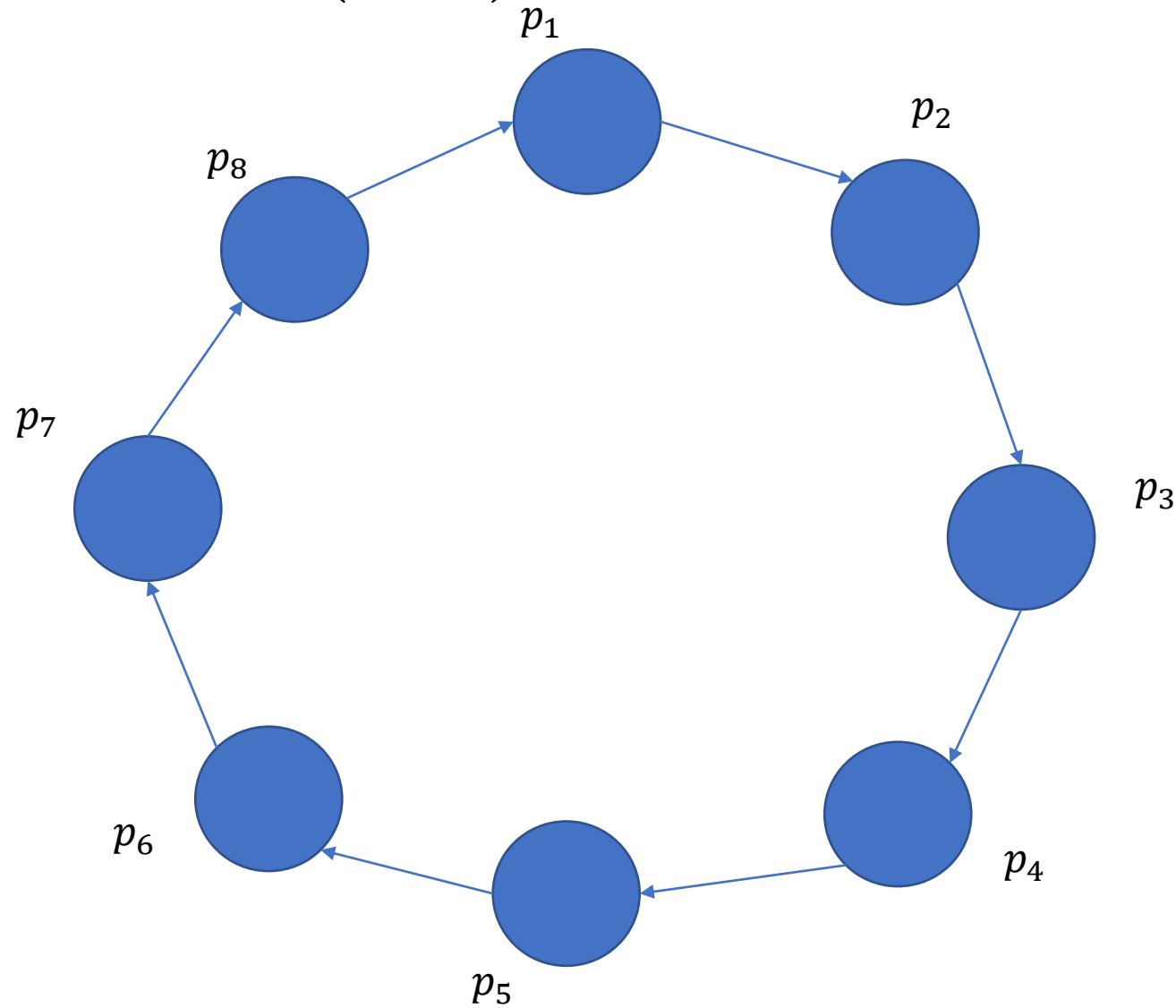
$$f_i(\{x_j(t) | j \in N_i\})$$

decide dacă la iterația curentă nodul i devine sau nu lider.

Asimptotic, soluția problemei este aducerea sistemului în starea:

$$x_i^* = \begin{cases} \text{lider,} & i = \operatorname{argmax}_j id_j \\ \text{non-lider,} & \text{altfel} \end{cases}$$

Alegere Lider (AL)

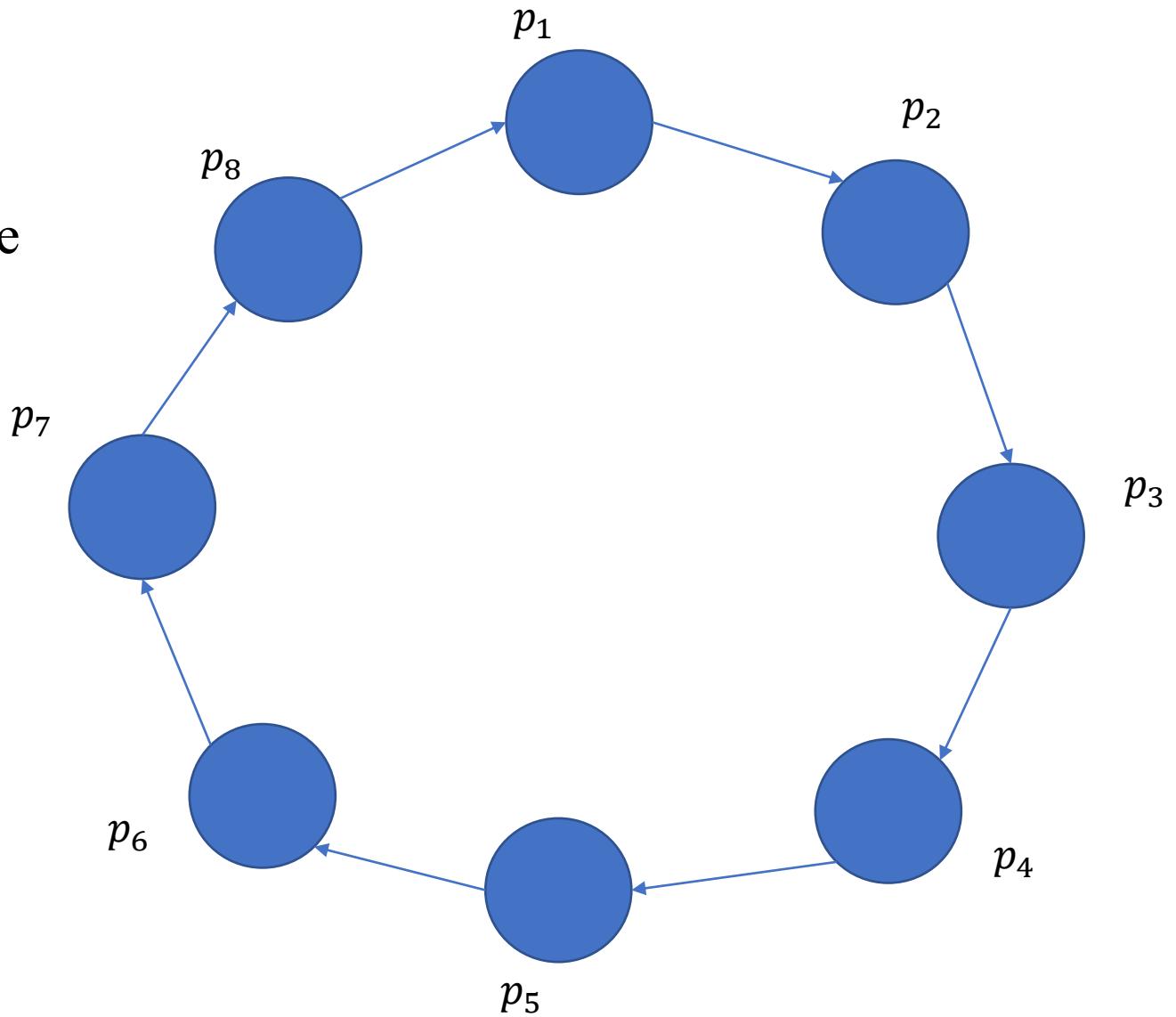


Alegere Lider (AL)

Ipoteze vedere locală:

1. Topologie inel unidirecțional (P_i cunoaște poziția relativă în inel)
2. Nodul P_i se identifică cu id_i
3. Nodul P_i cunoaște nr. de noduri n

Problema se reduce la: *specifică un program SPMD (f_i) astfel încât, într-un număr minim de iterații să asigurăm convergența stării globale la starea optimă, i.e. $x_i(T) = x_i^*$.*



Alegere Lider (coordonate globale)

Algoritm **AlegeLiderInel_cg()**:

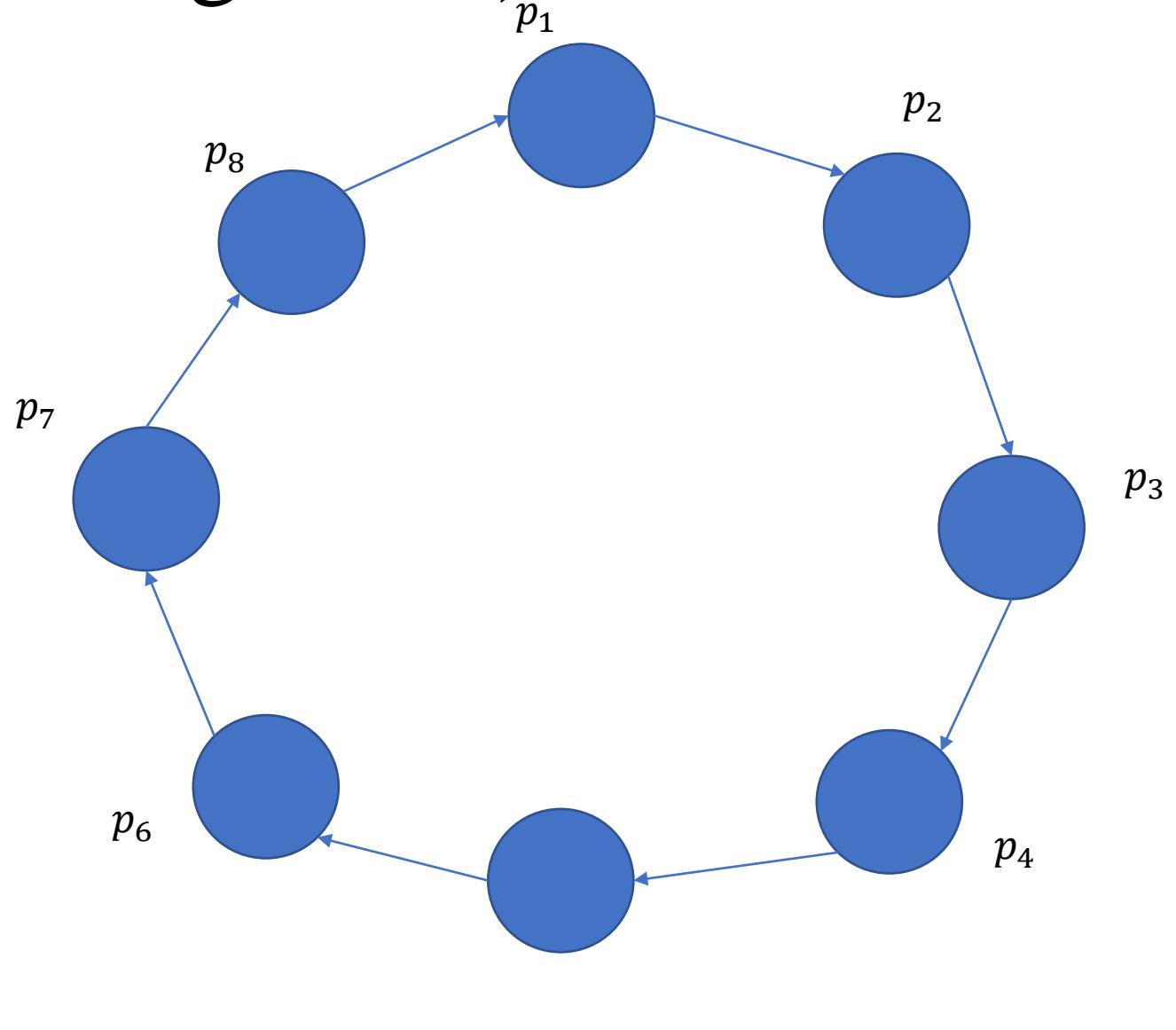
M_i : - int n (număr noduri)
- int i (index propriu)
- int id (id propriu)
- int id_max (id propriu)

% Faza I: max ID

- Calculează $\max\{id_1, id_2, \dots, id_n\}$
- Rezultatul va fi stocat într-un nod particular

% Faza II: Difuzare Max ID (Broadcast)

- Rezultatul este difuzat peste tot inelul
- Stările x_i sunt ajustate conform rezultatului



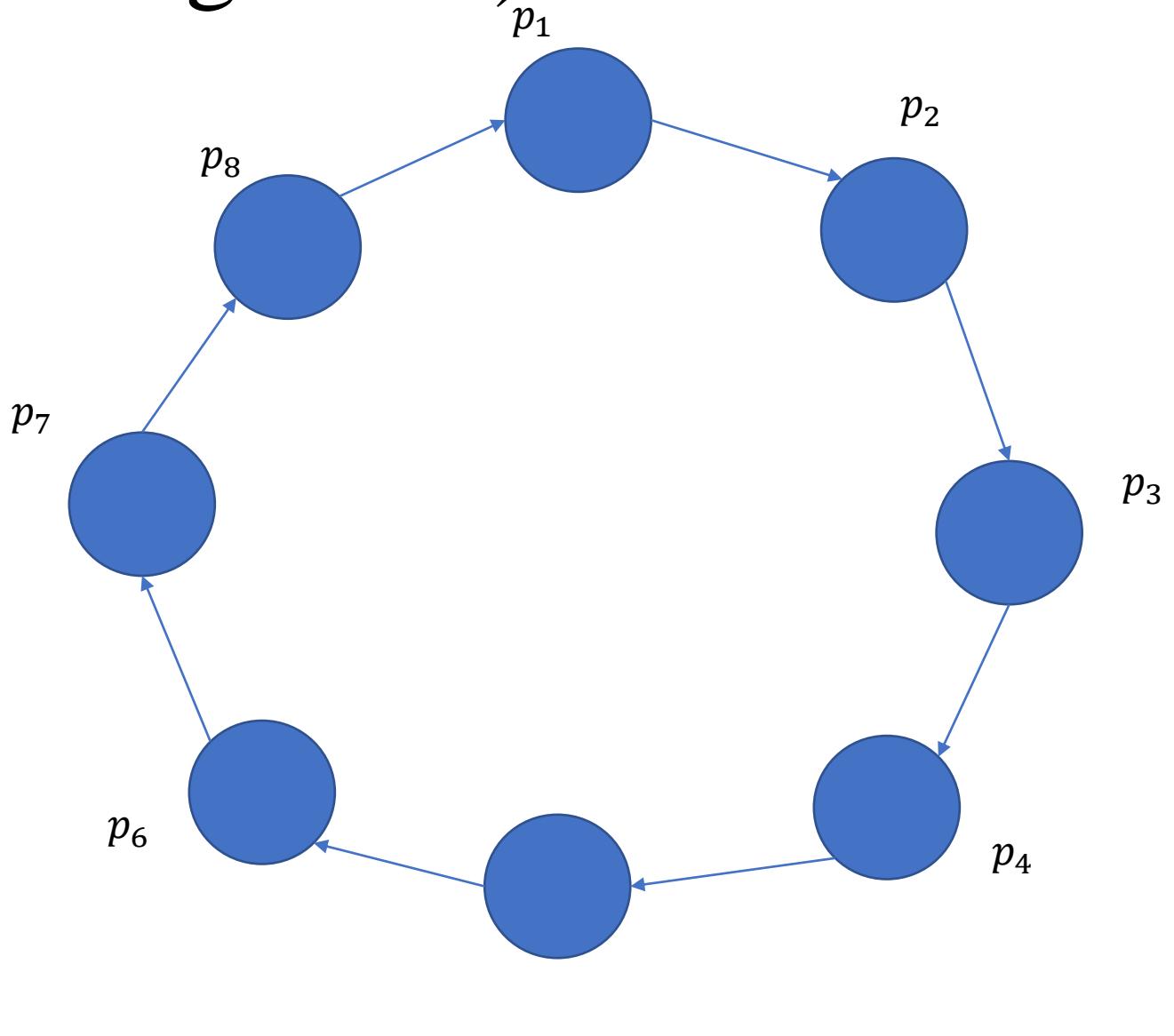
Alegere Lider (coordonate globale)

M_i : - int n (număr noduri)
- int i (index propriu)
- int id (id propriu)
- int id_max (id propriu)

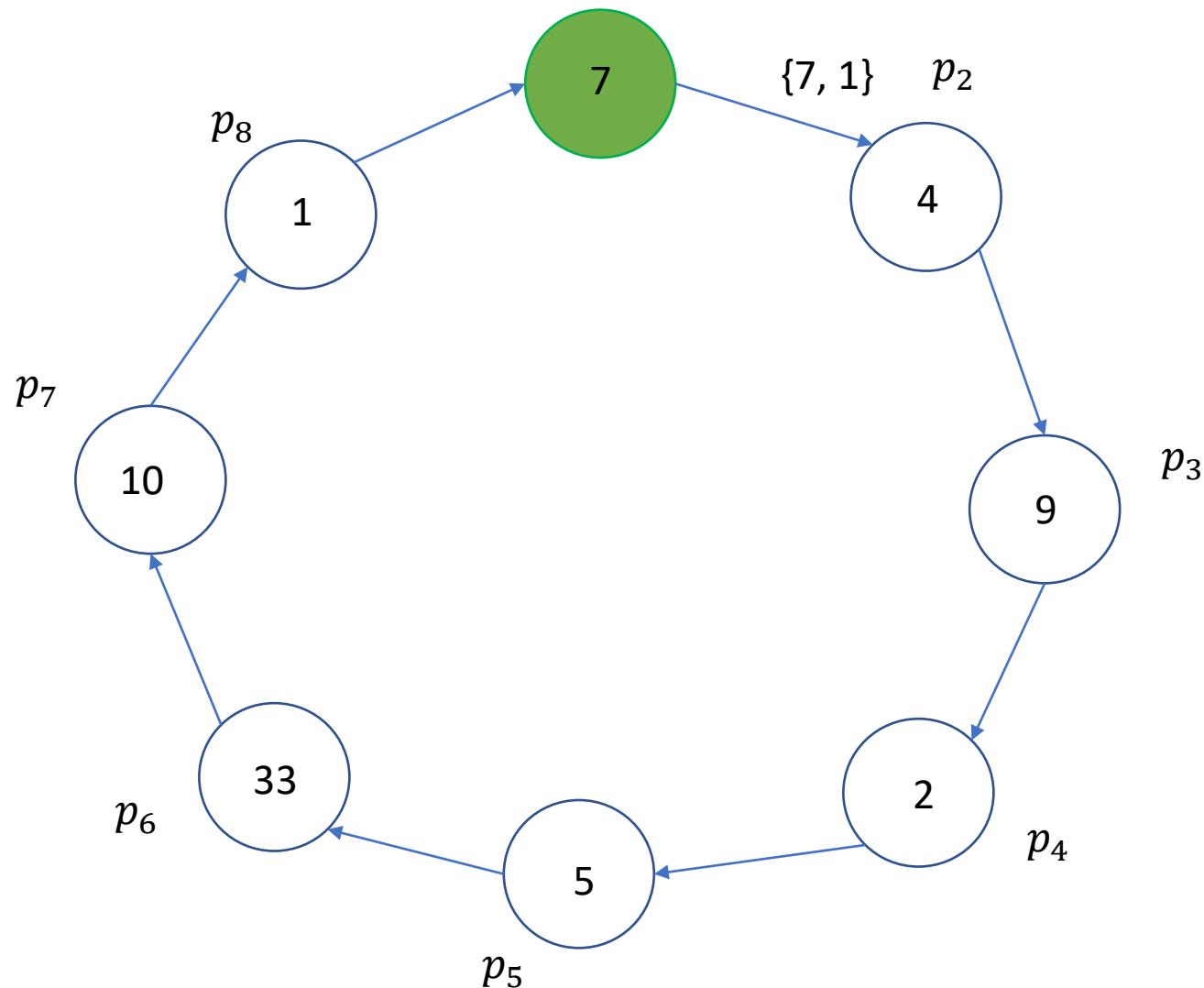
% Faza max ID

Functie transformare nod i $f_i()$:

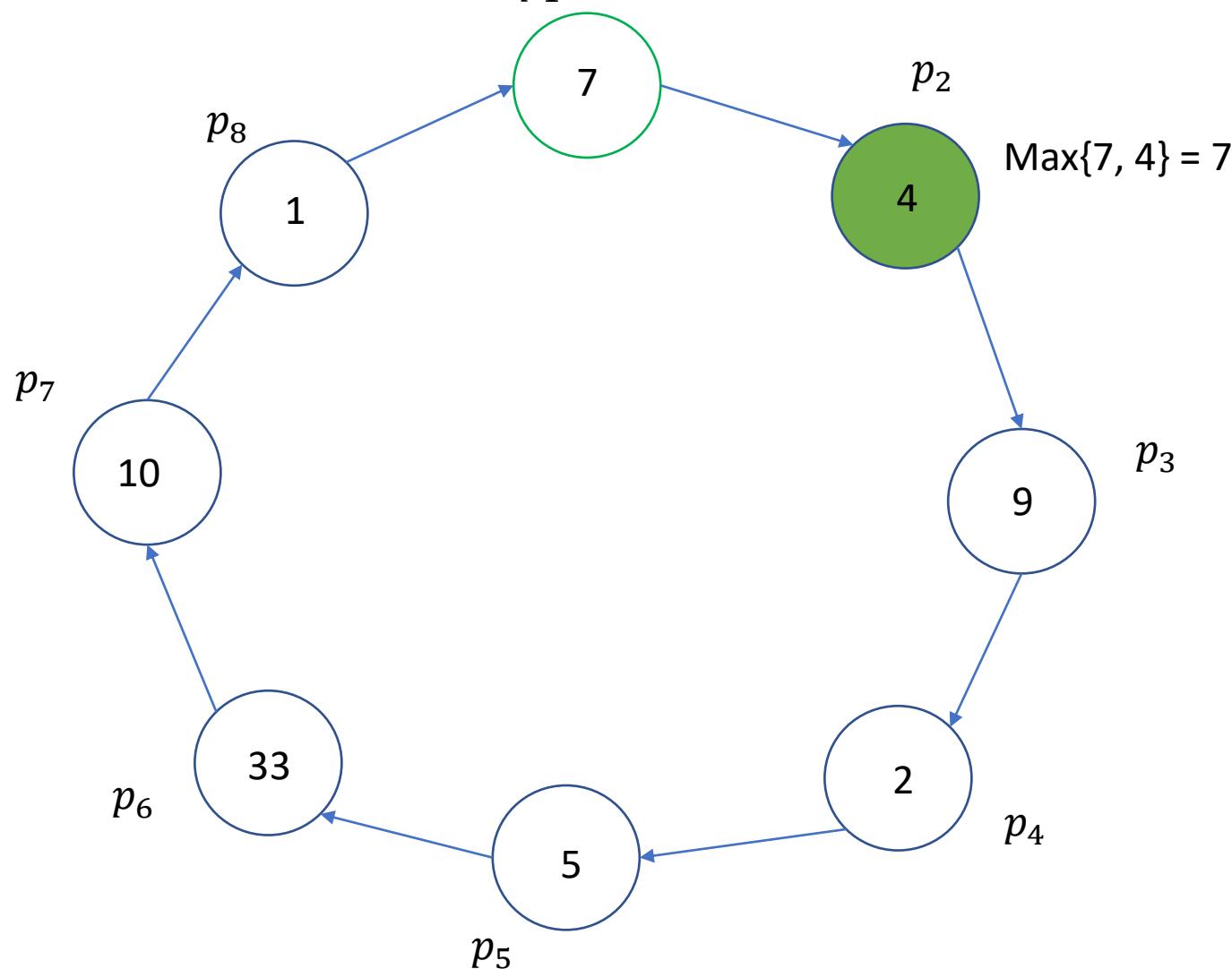
1. **If (i == 1):**
 1. send(id, 2);
 2. id_aux = recv(n);
2. **else:**
 1. id_aux = recv(index - 1 mod n);
 2. send(idmax, index + 1 mod n);
3. idmax = max(id, id_aux);



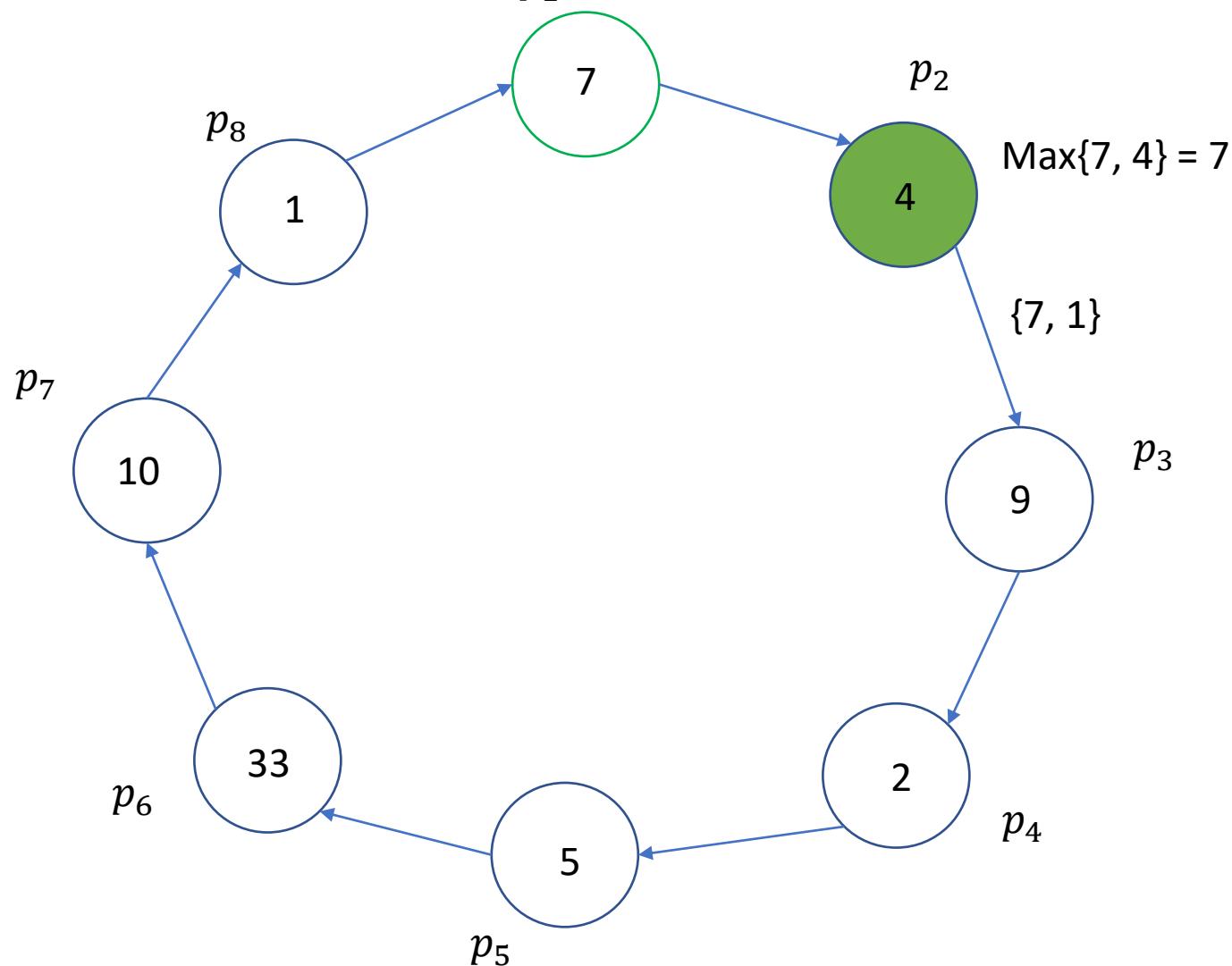
Alegere Lider (coordonate globale)



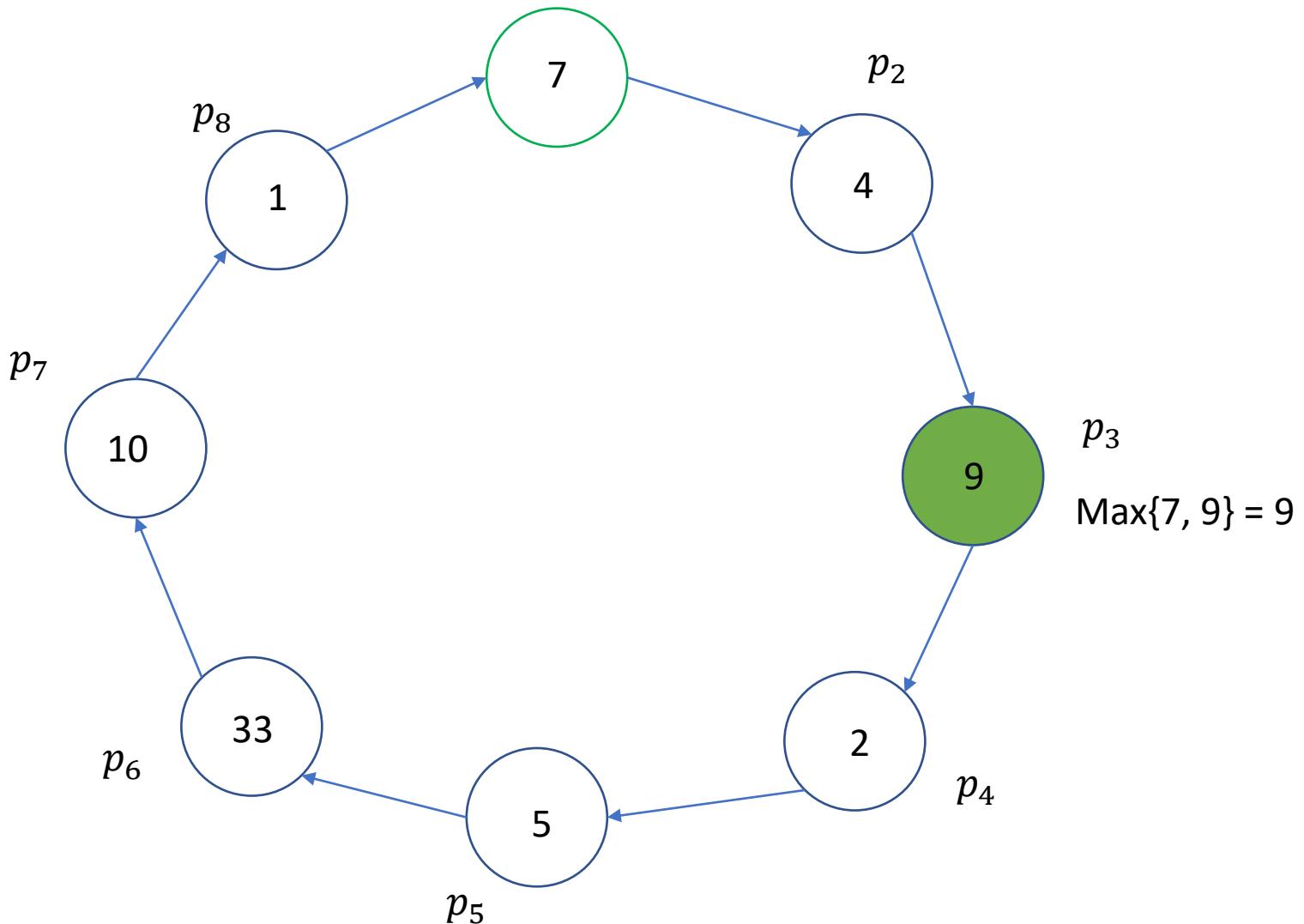
Alegere Lider (coordonate globale)



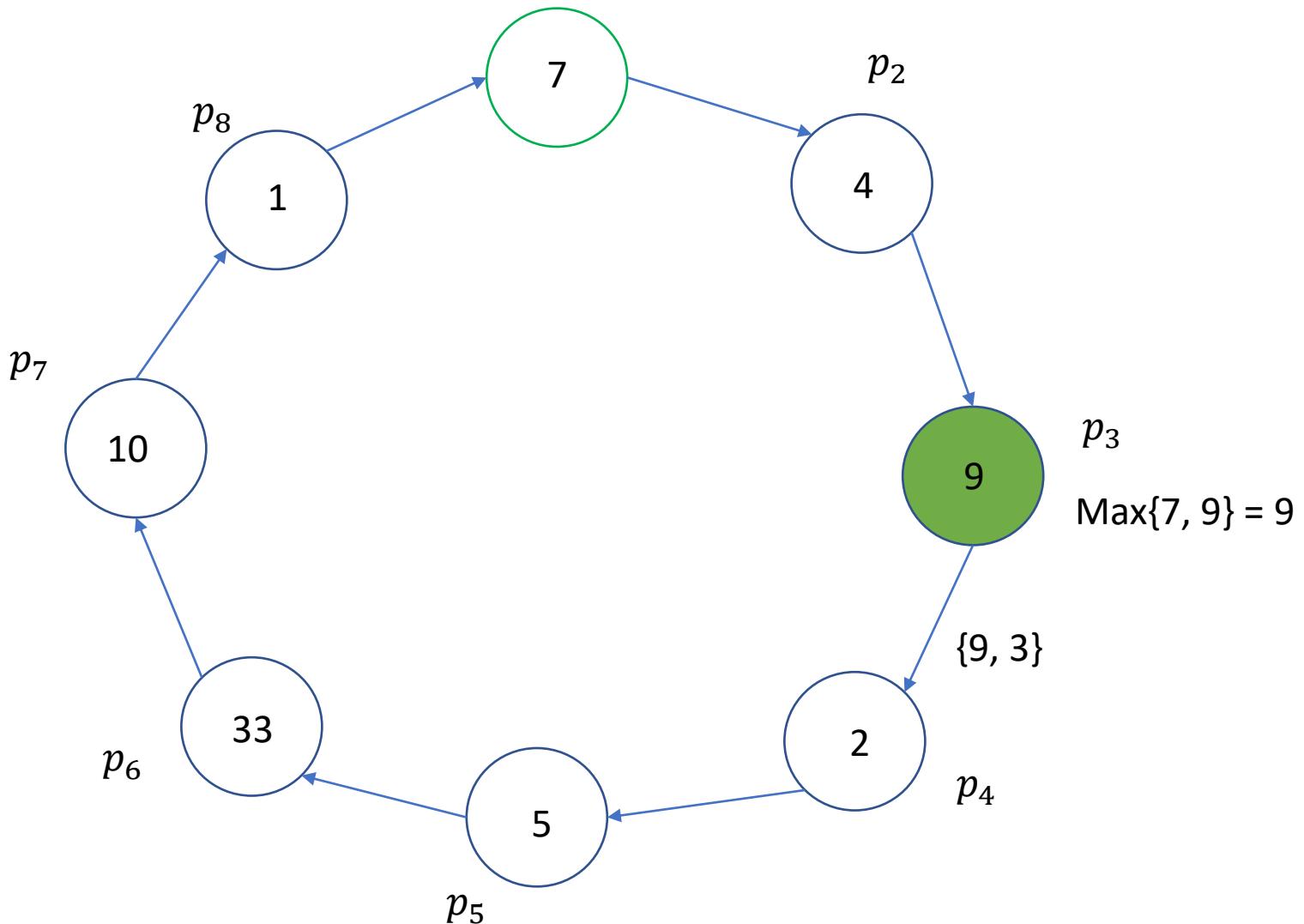
Alegere Lider (coordonate globale)



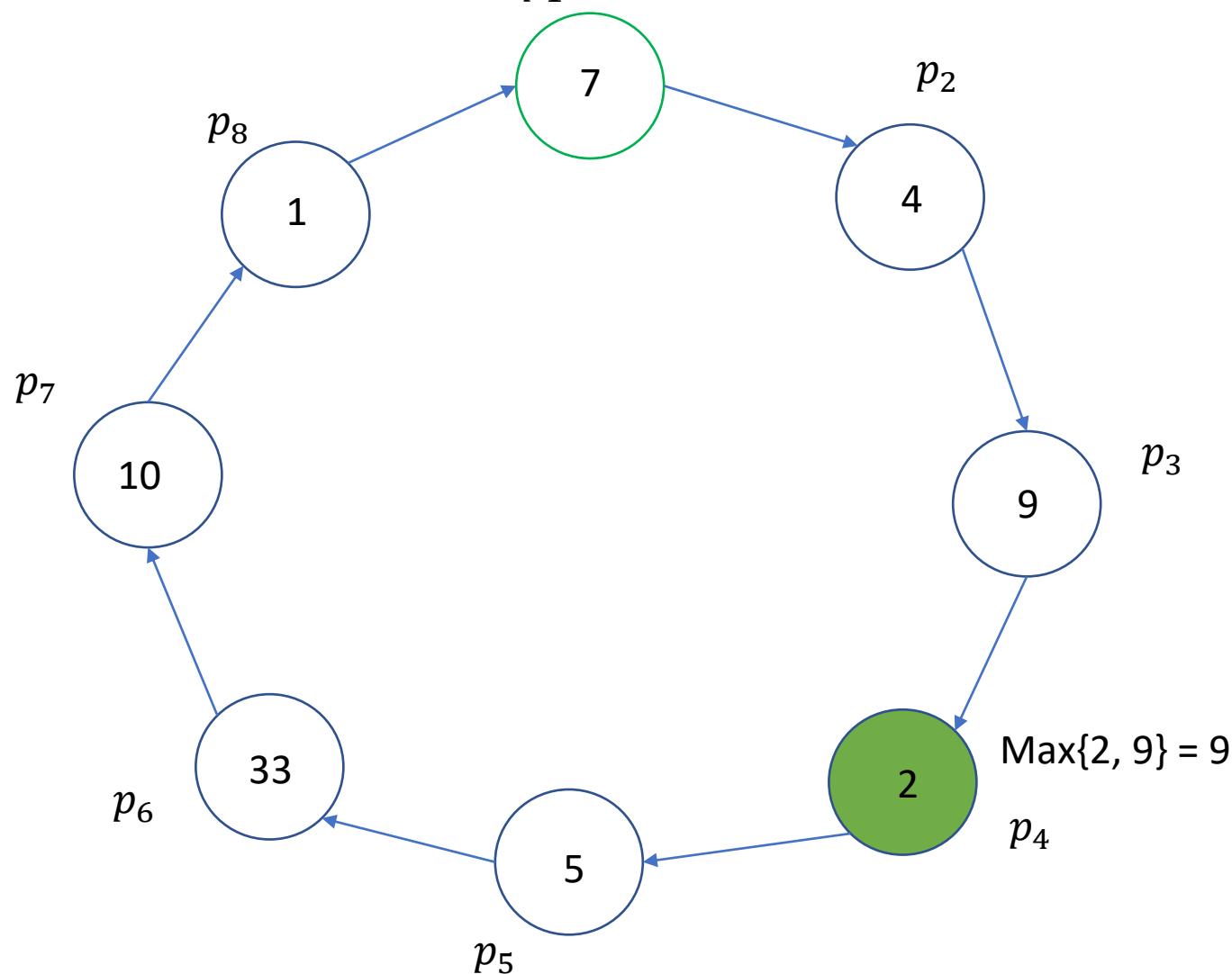
Alegere Lider (coordonate globale)



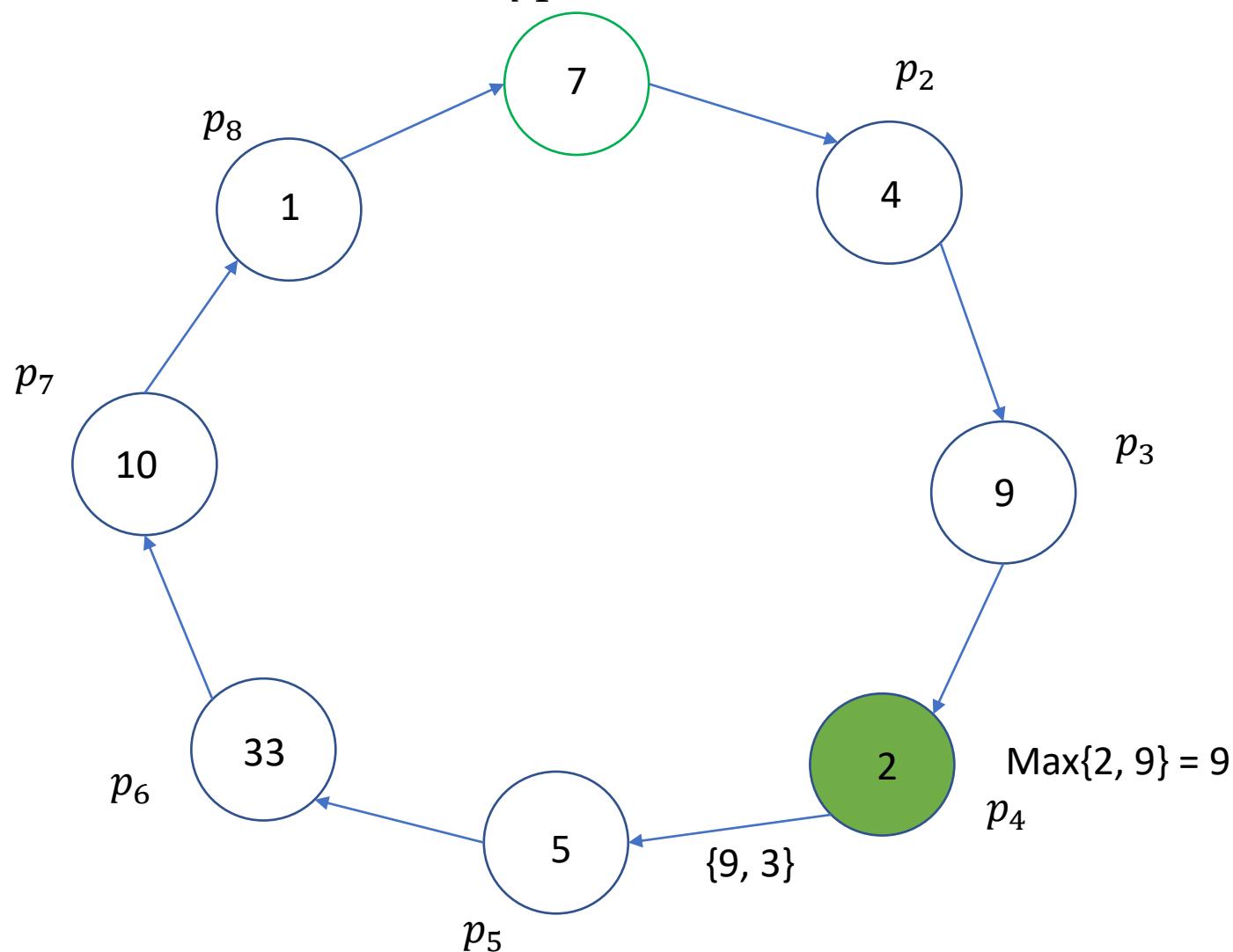
Alegere Lider (coordonate globale)



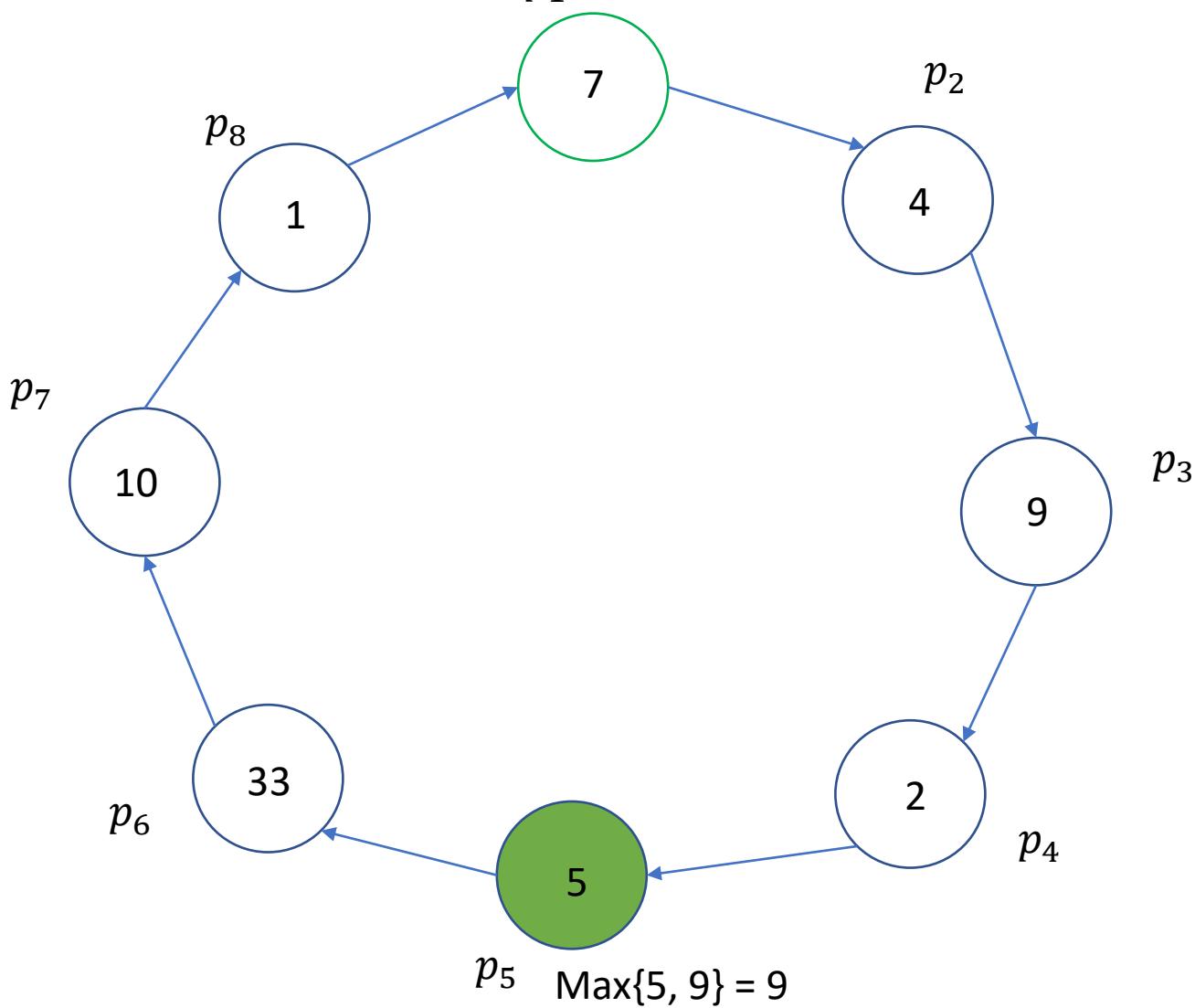
Alegere Lider (coordonate globale)



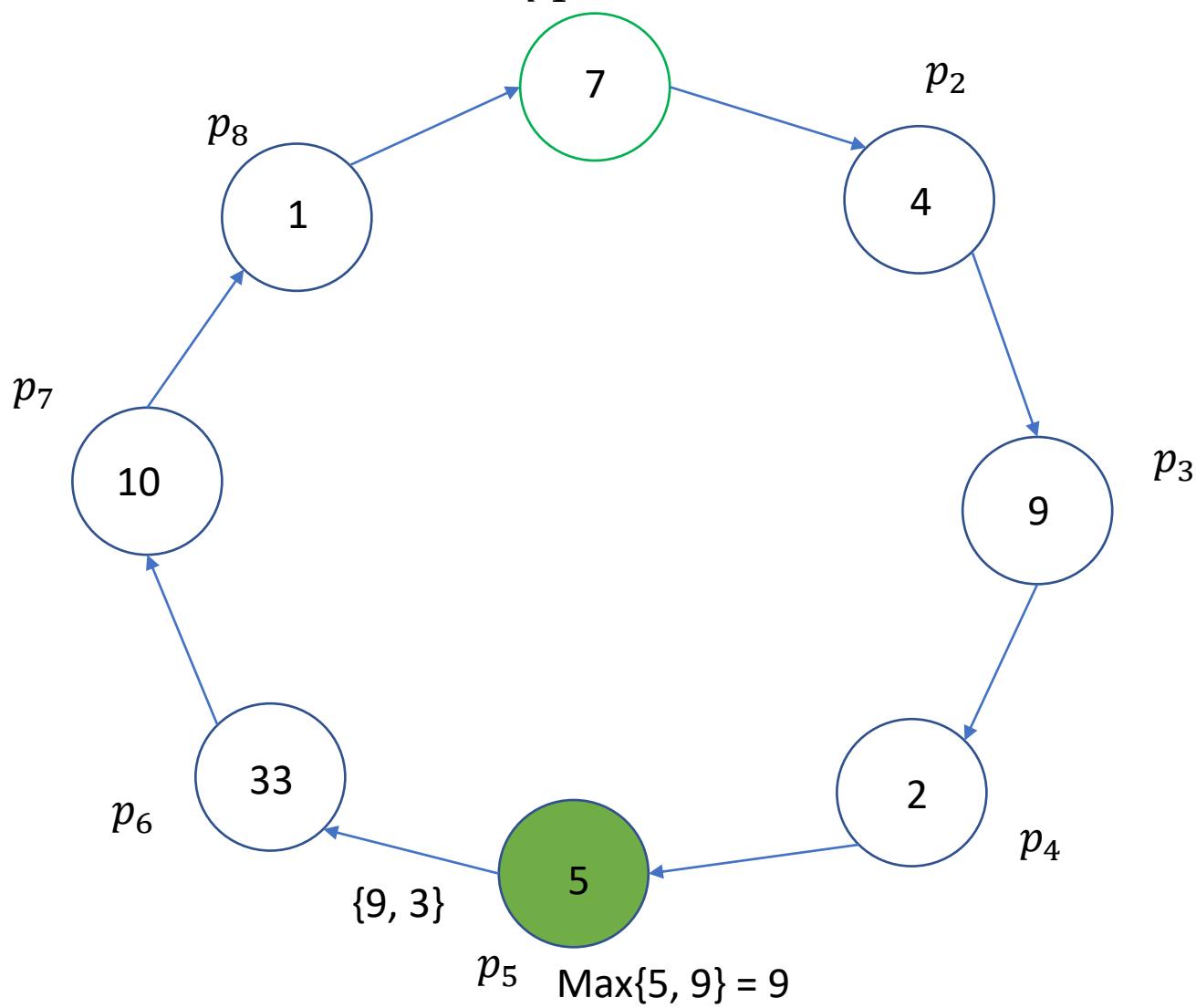
Alegere Lider (coordonate globale)



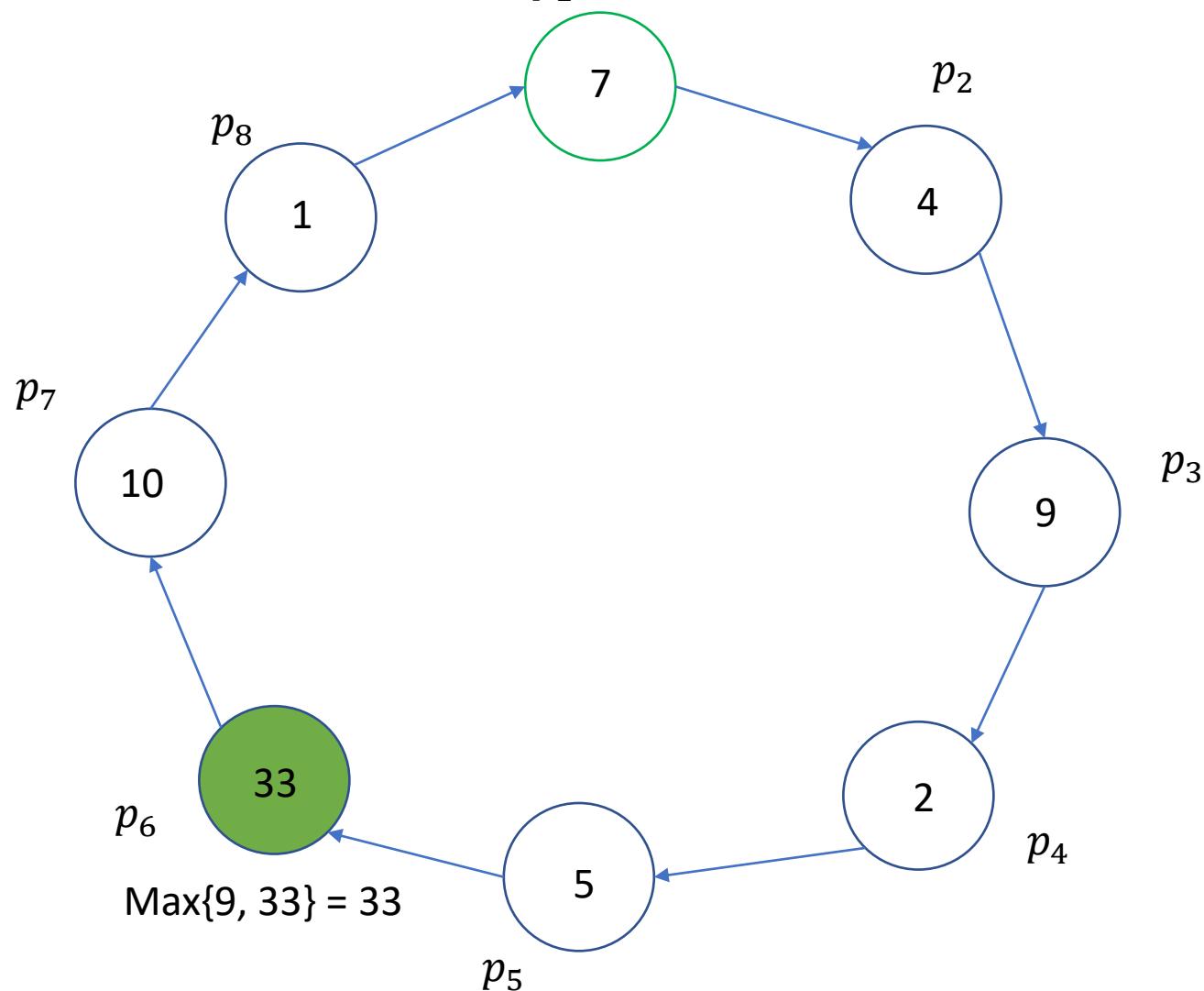
Alegere Lider (coordonate globale)



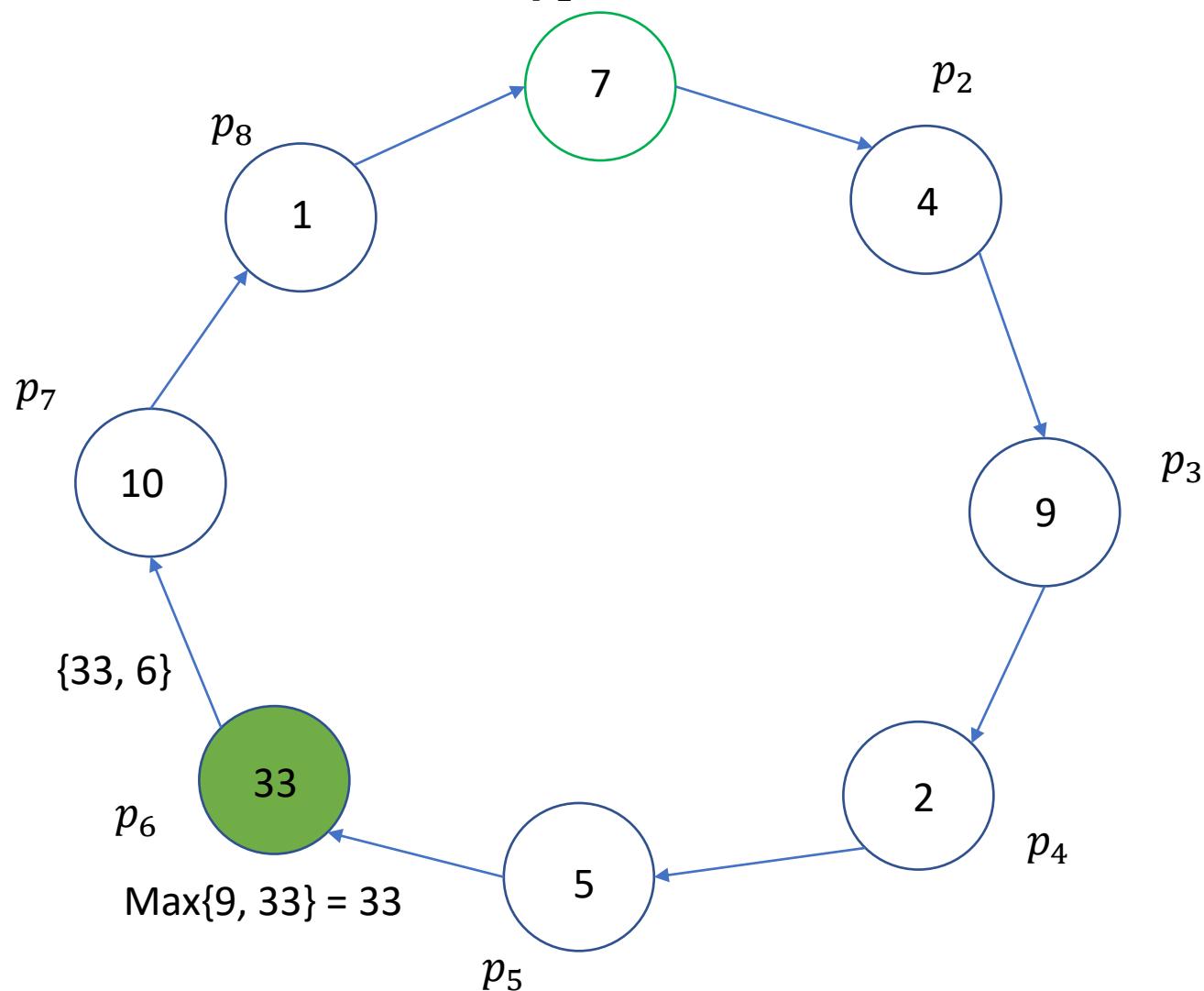
Alegere Lider (coordonate globale)



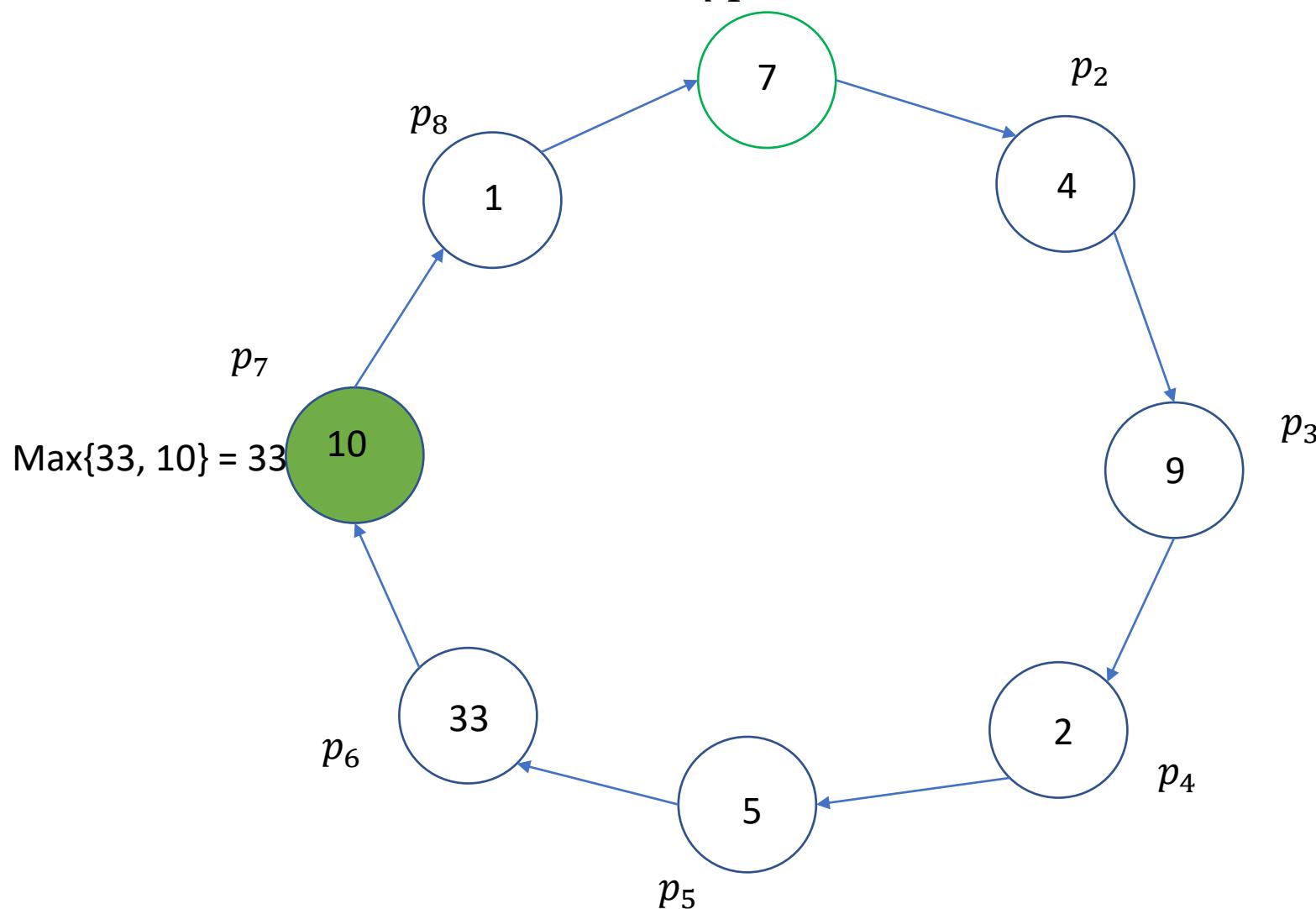
Alegere Lider (coordonate globale)



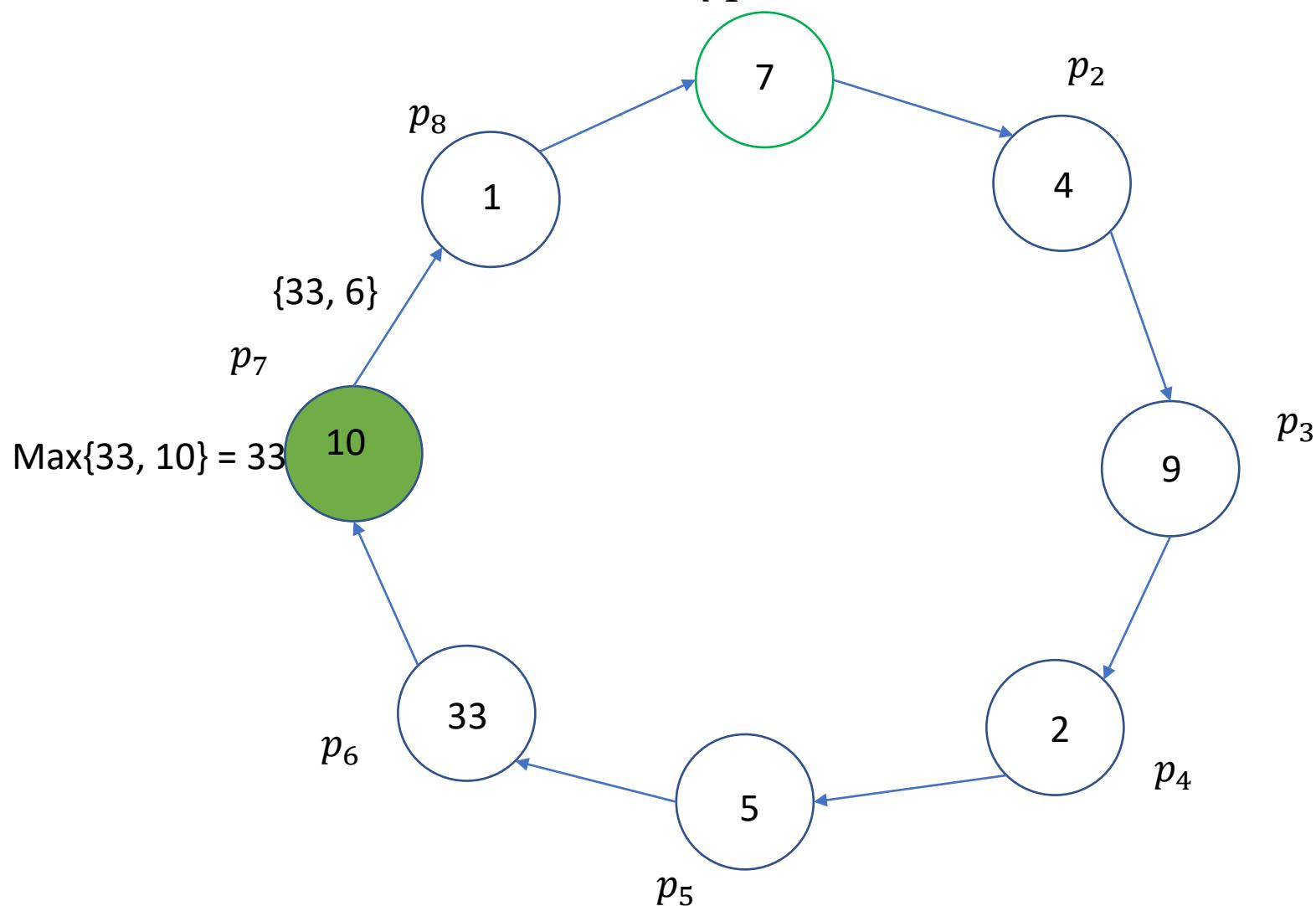
Alegere Lider (coordonate globale)



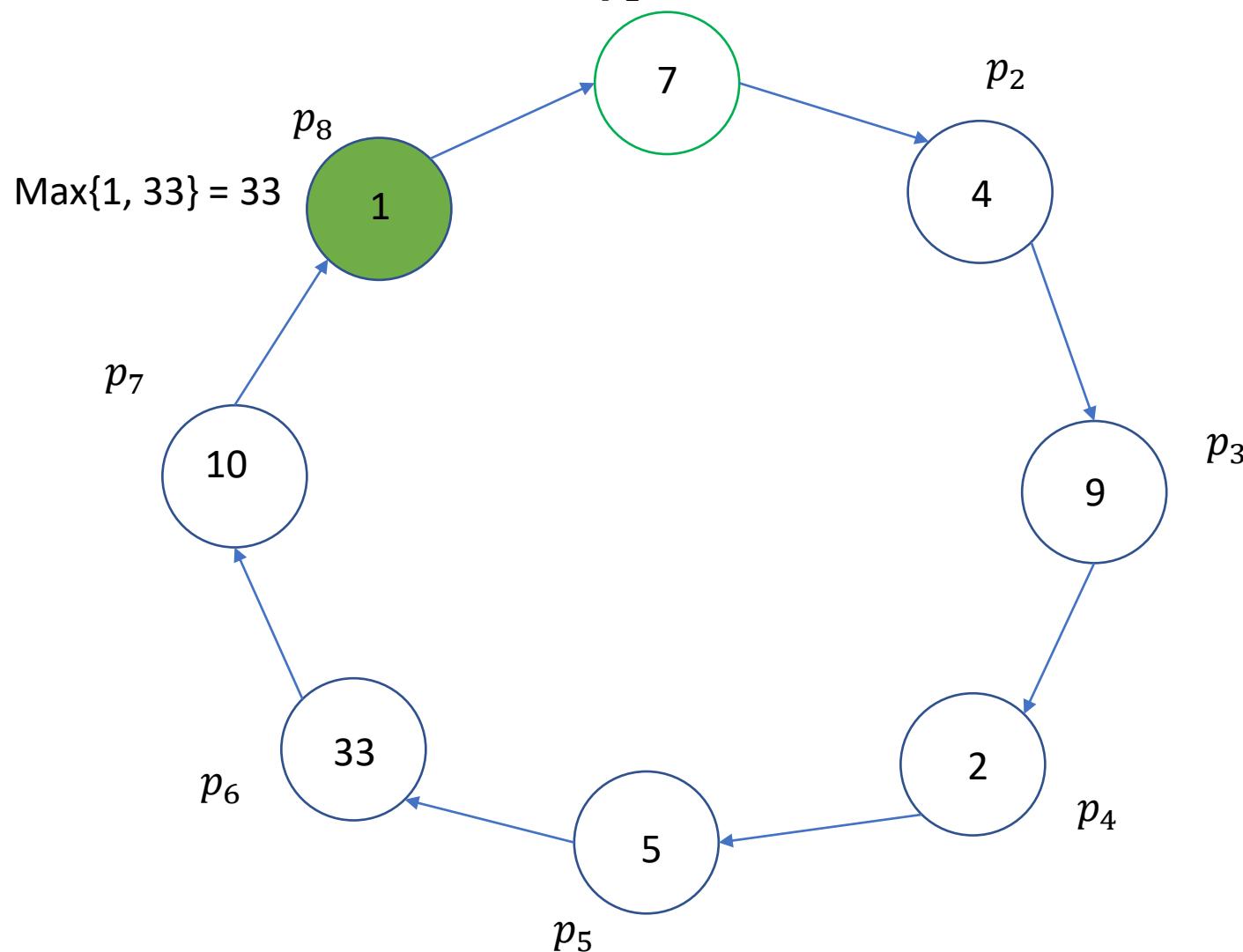
Alegere Lider (coordonate globale)



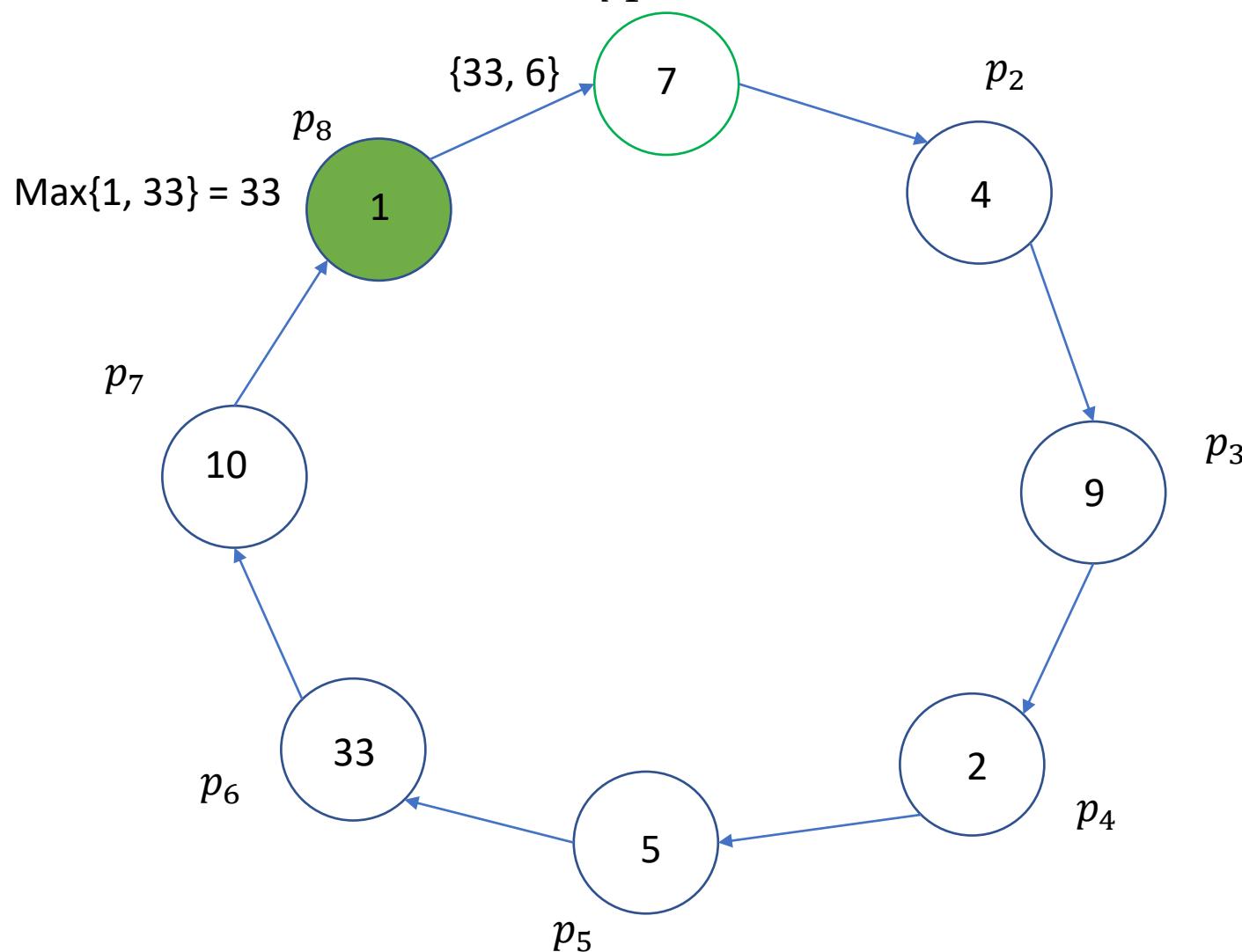
Alegere Lider (coordonate globale)



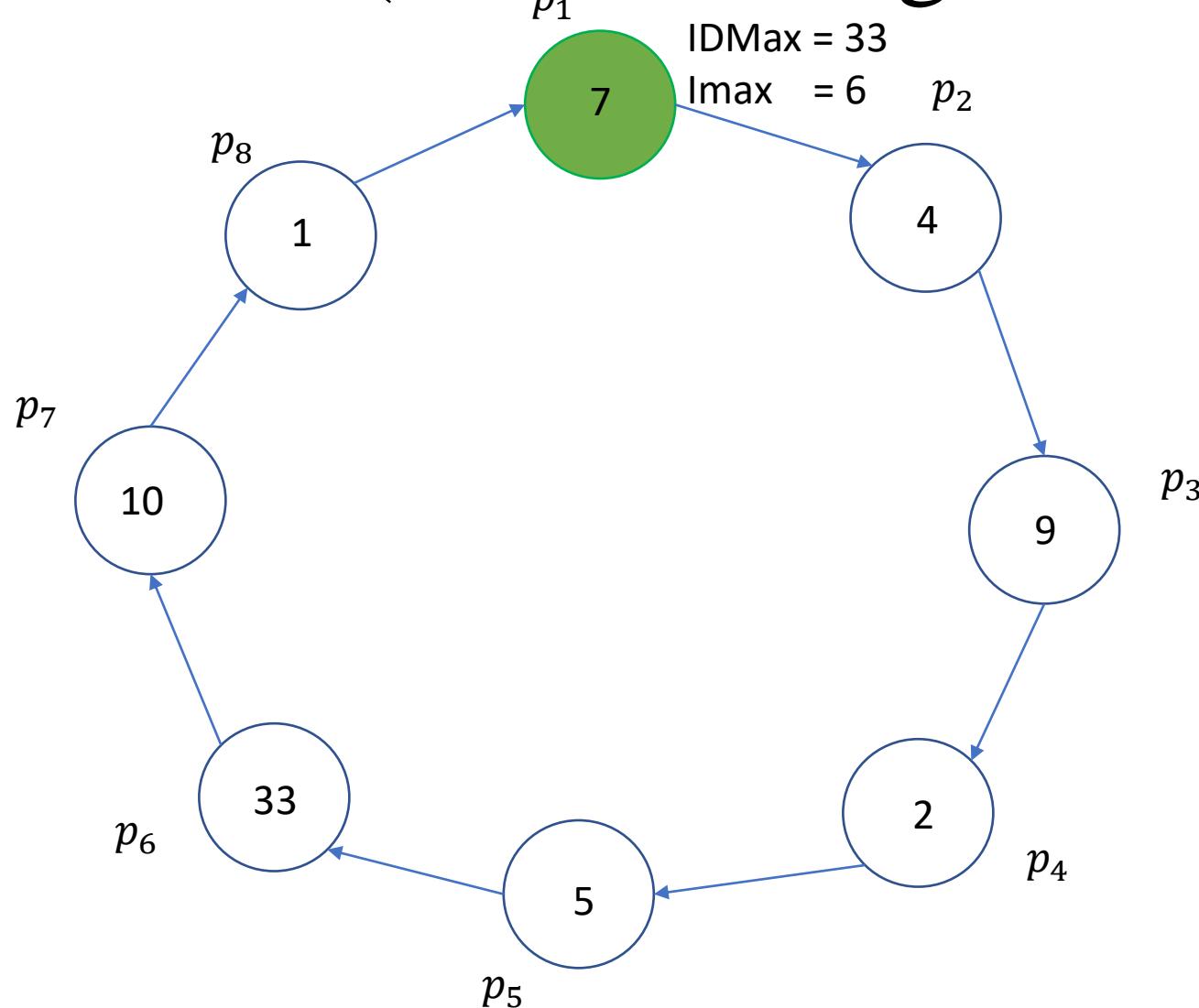
Alegere Lider (coordonate globale)



Alegere Lider (coordonate globale)



Alegere Lider (coordonate globale)



Alegere Lider (coordonate globale)

Memorie locală nod i:

- int n (număr noduri)
- int i (index propriu)
- int id (id propriu)
- int id_max (id propriu)

% Faza I: Max ID

...

% Faza II: Difuzare Max ID (Broadcast)

Functie transformare nod i $f_i()$:

1. If (i == 1):

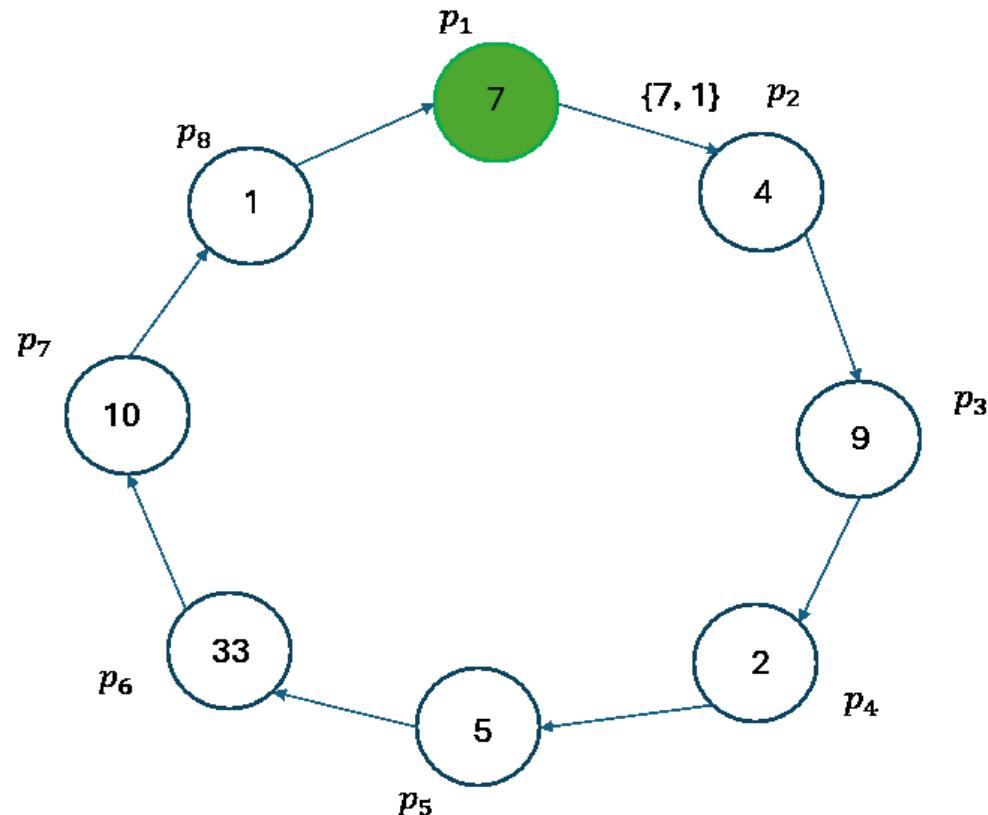
1. send({idmax, imax}, 2);

Else If (i == n-1):

1. {idmax, imax} = recv(index - 1 mod n);

Else

1. {idmax, imax} = recv(index - 1 mod n);
2. send({idmax, imax}, index + 1 mod n);

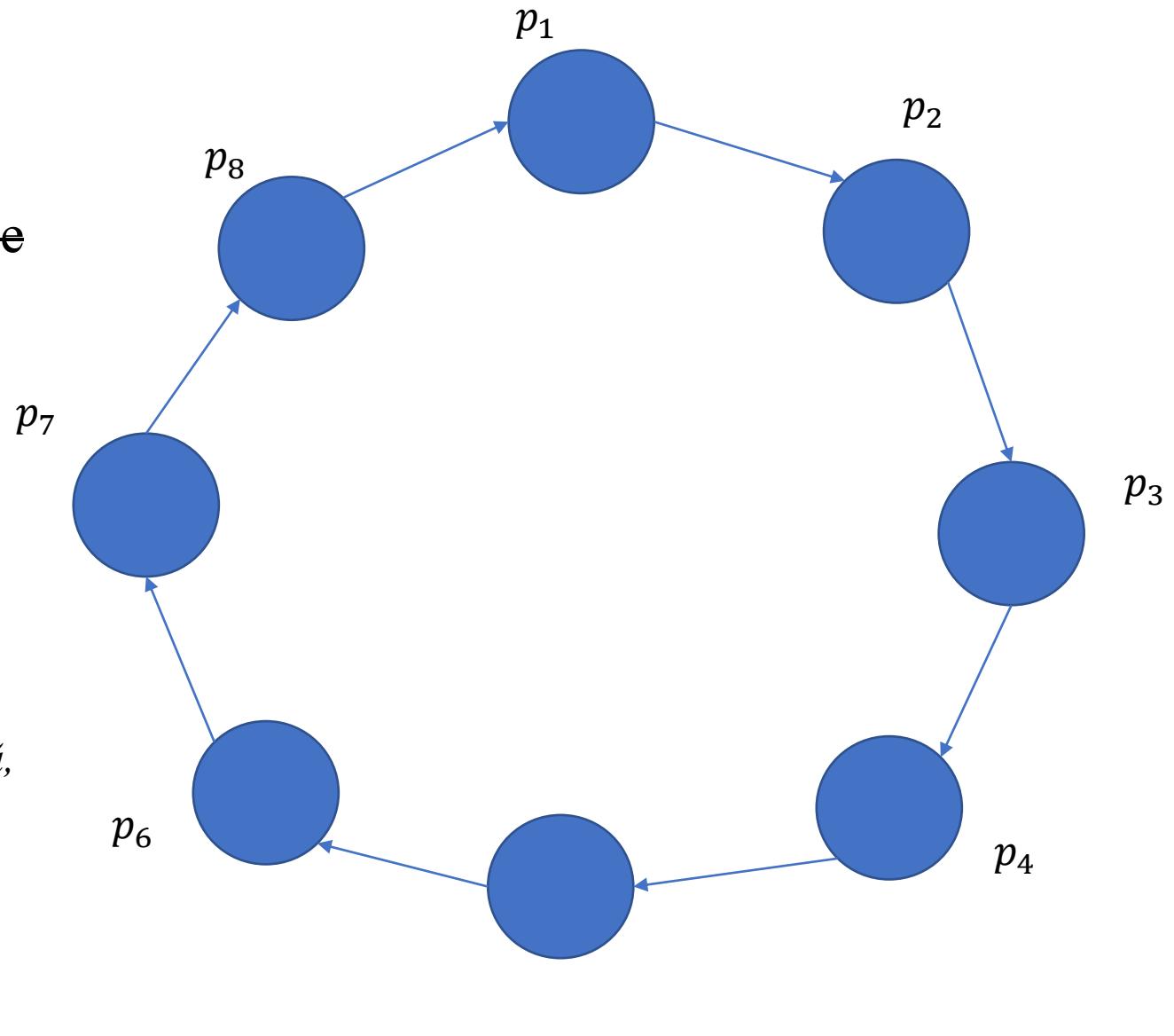


Alegere Lider (AL)

Ipoteze vedere locală:

1. Topologie inel unidirecțional (P_i cunoaște poziția relativă în inel)
2. Nodul P_i se identifică cu id_i
3. ~~Nodul P_i cunoaște nr. de noduri n~~

- Ipoteze mai realiste (rețele peer-to-peer)
- Topologia și numărul de noduri sunt statice
- Problema se reduce la: *specifică un program SPMD (f_i) astfel încât, într-un număr minim de iterații să asigurăm convergența stării globale la starea optimă, i.e. $x_i(T) = x_i^*$.*



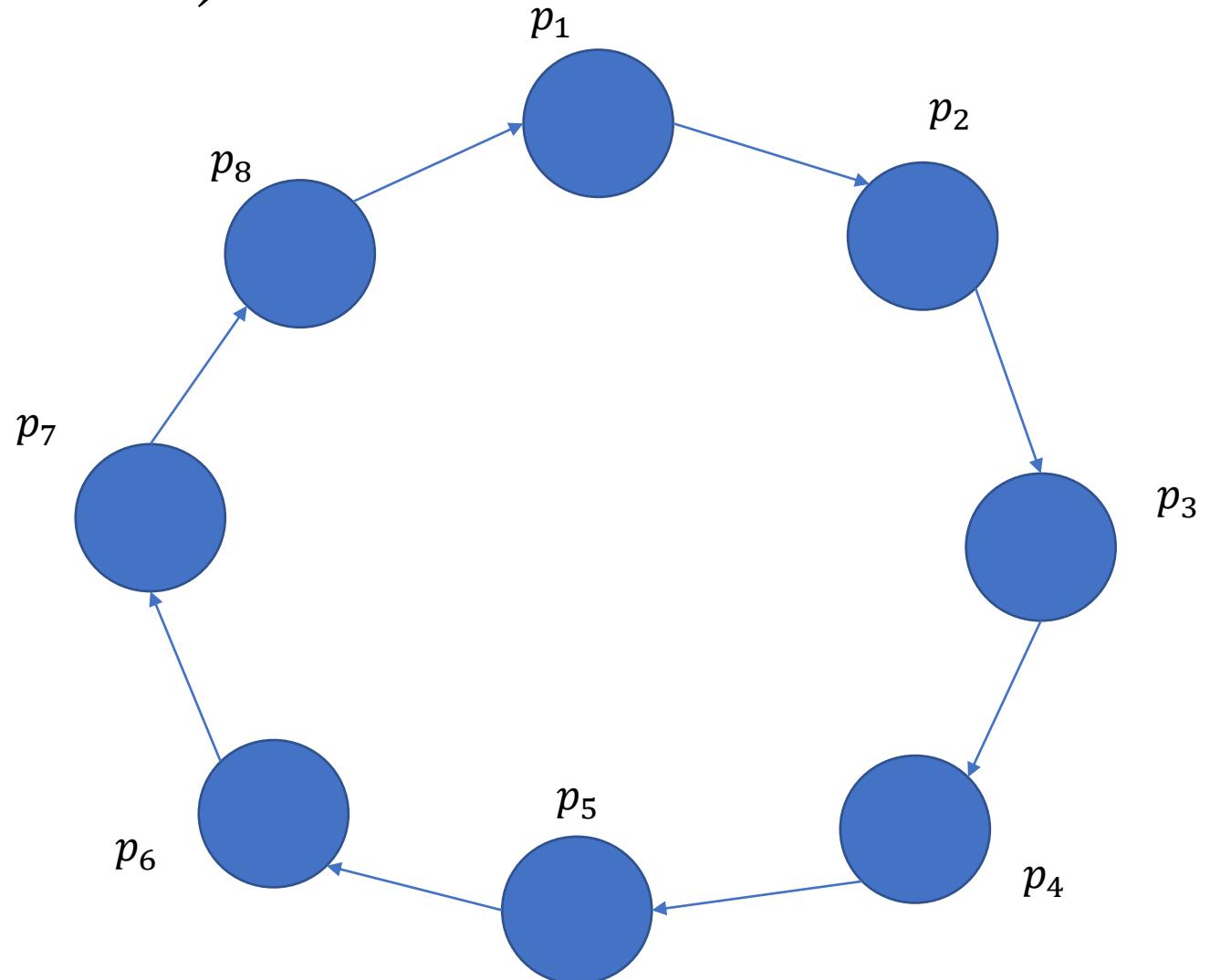
Algoritmul Flooding (LCR)

Algoritm **Flooding()**:

M_i : - int id (id propriu)
- int send_id (var auxiliară), inițial id
- status $\in \{\text{lider}, \text{non-lider}\}$, inițial non-lider

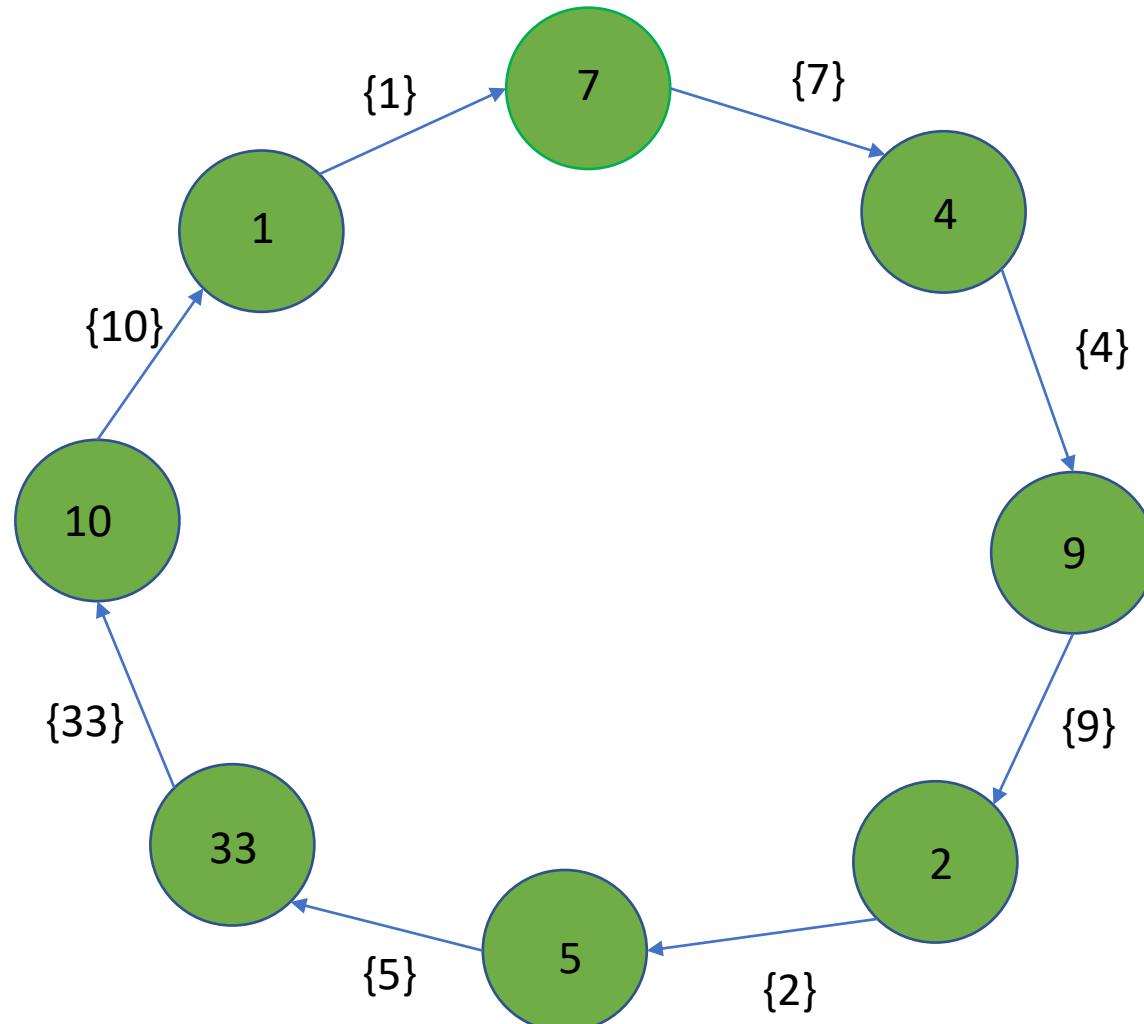
Functie transformare nod i ():

1. send(send_id, index + 1 mod n);
2. recv_id = recv(index - 1 mod n);
3. **If** (recv_id > id):
 1. send_id := recv_id;
4. **ElseIf** (recv_id == id): status = leader;

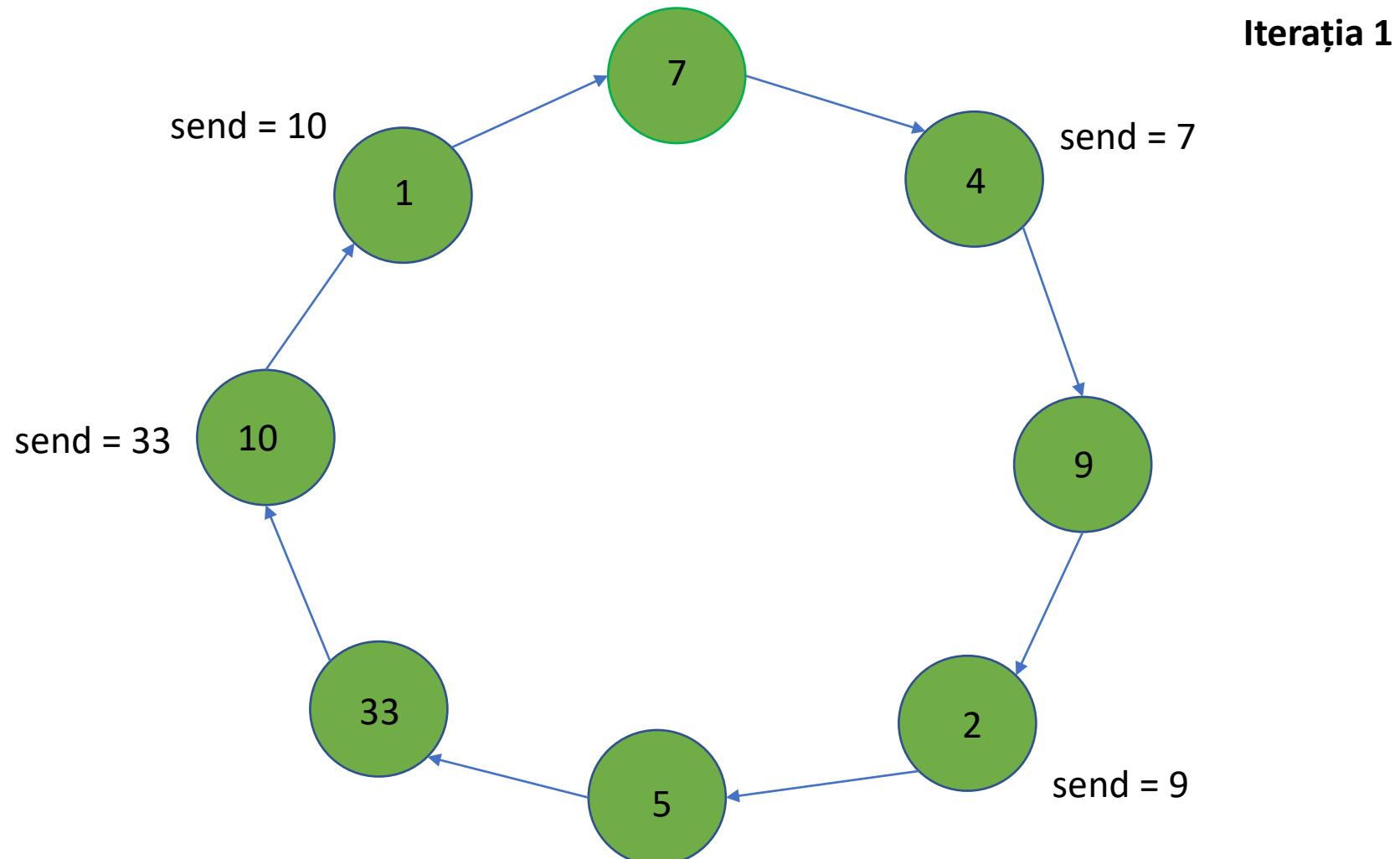


Algoritmul Flooding (LCR)

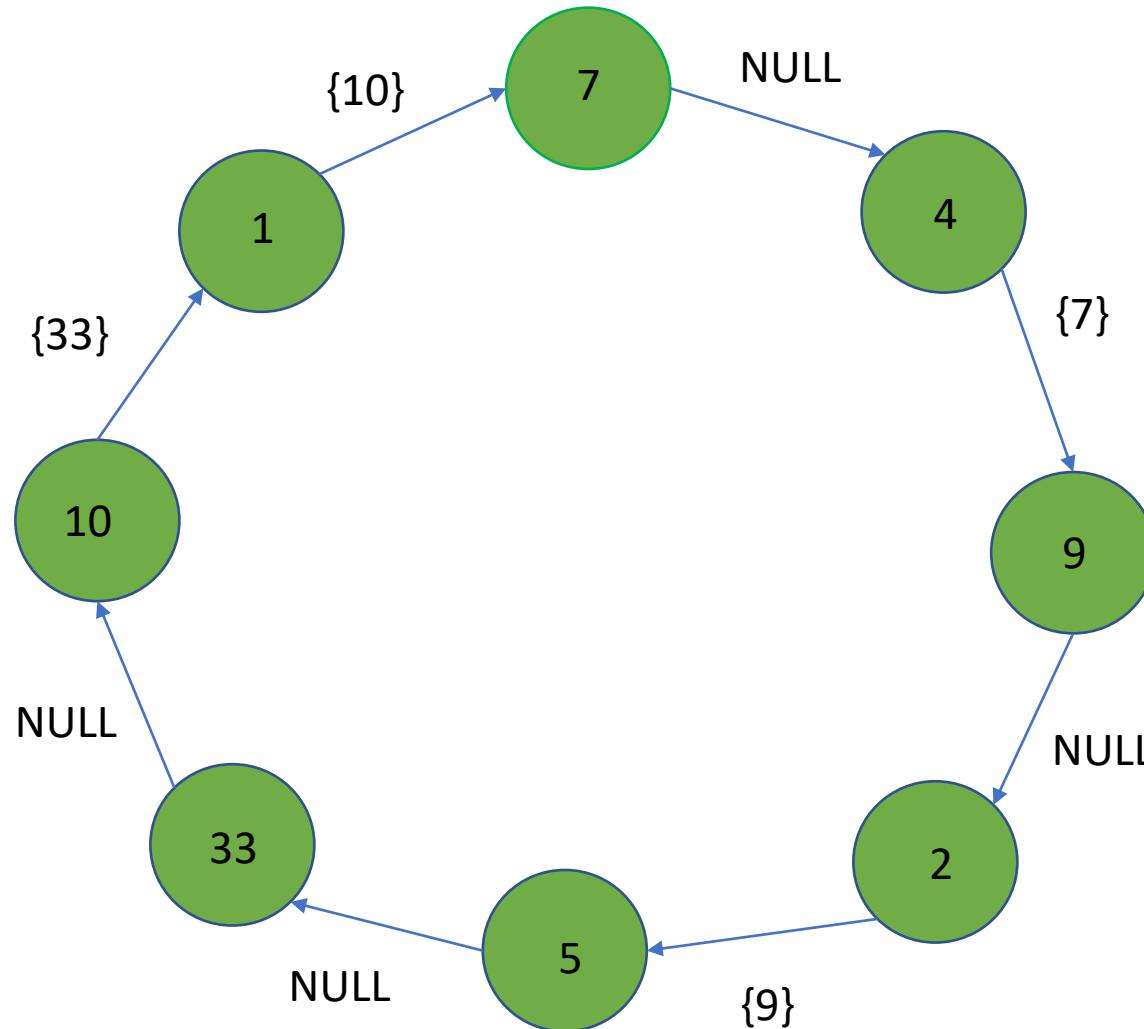
Iterația 1



Algoritmul Flooding (LCR)

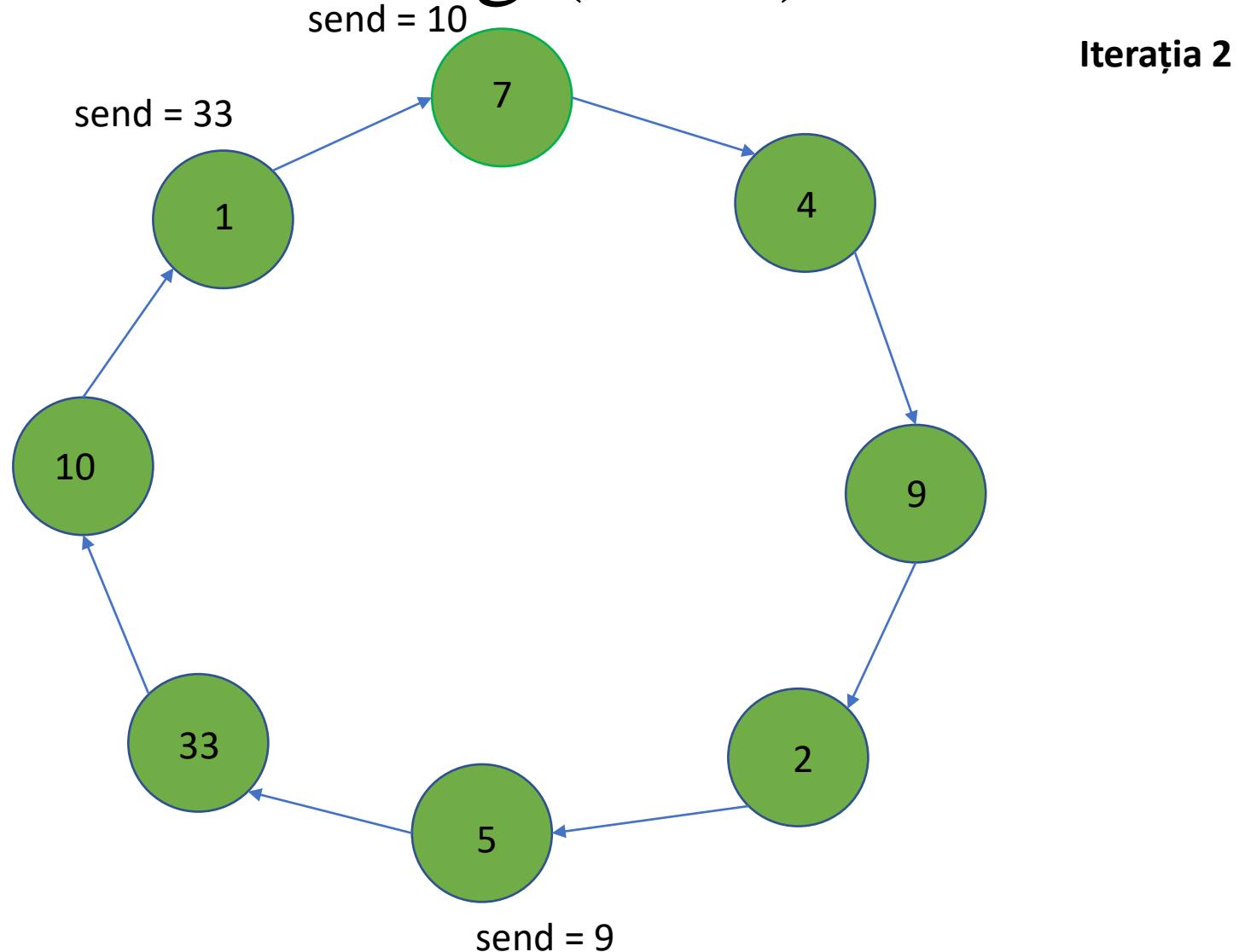


Algoritmul Flooding (LCR)



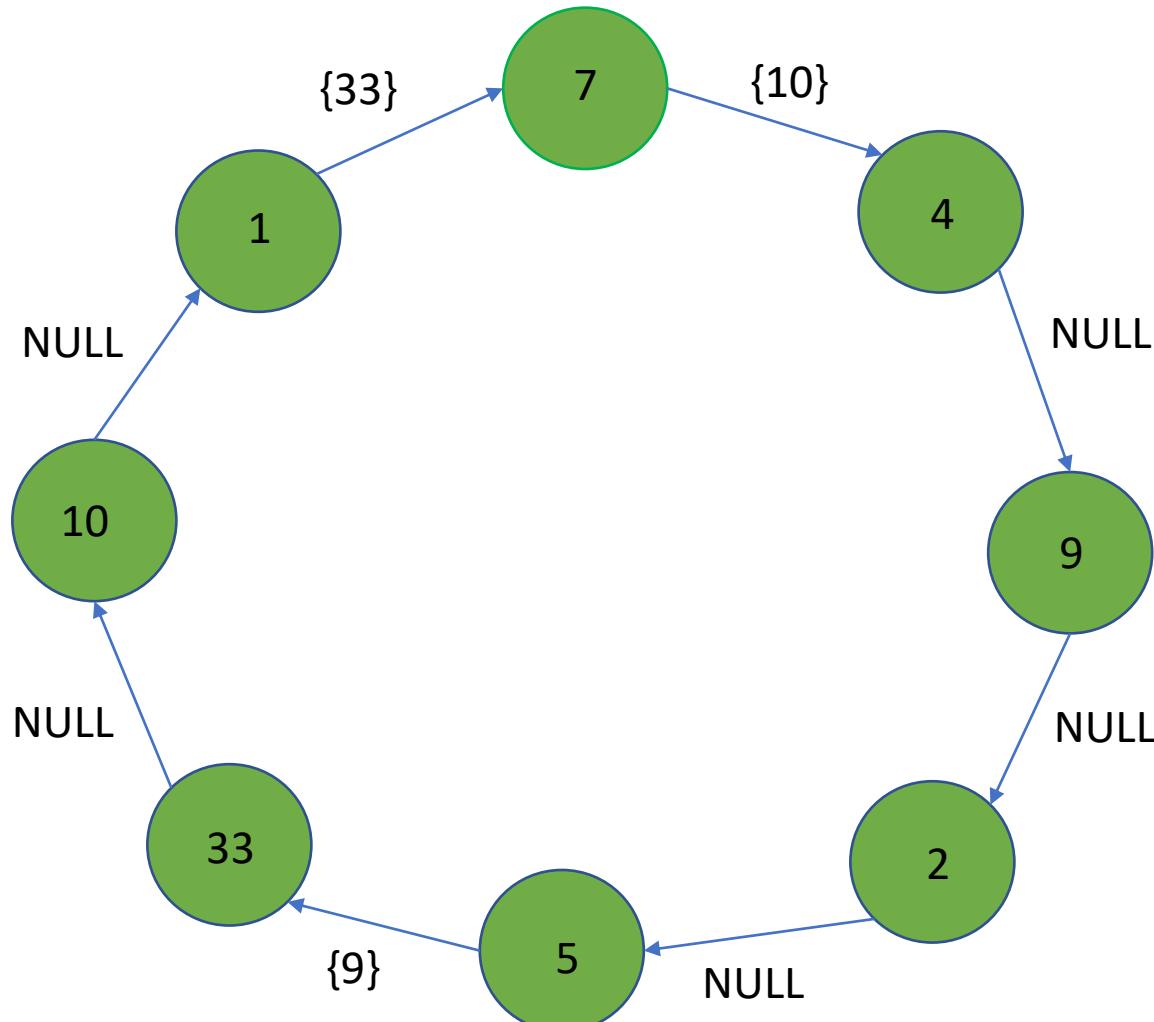
Iterația 2

Algoritmul Flooding (LCR)

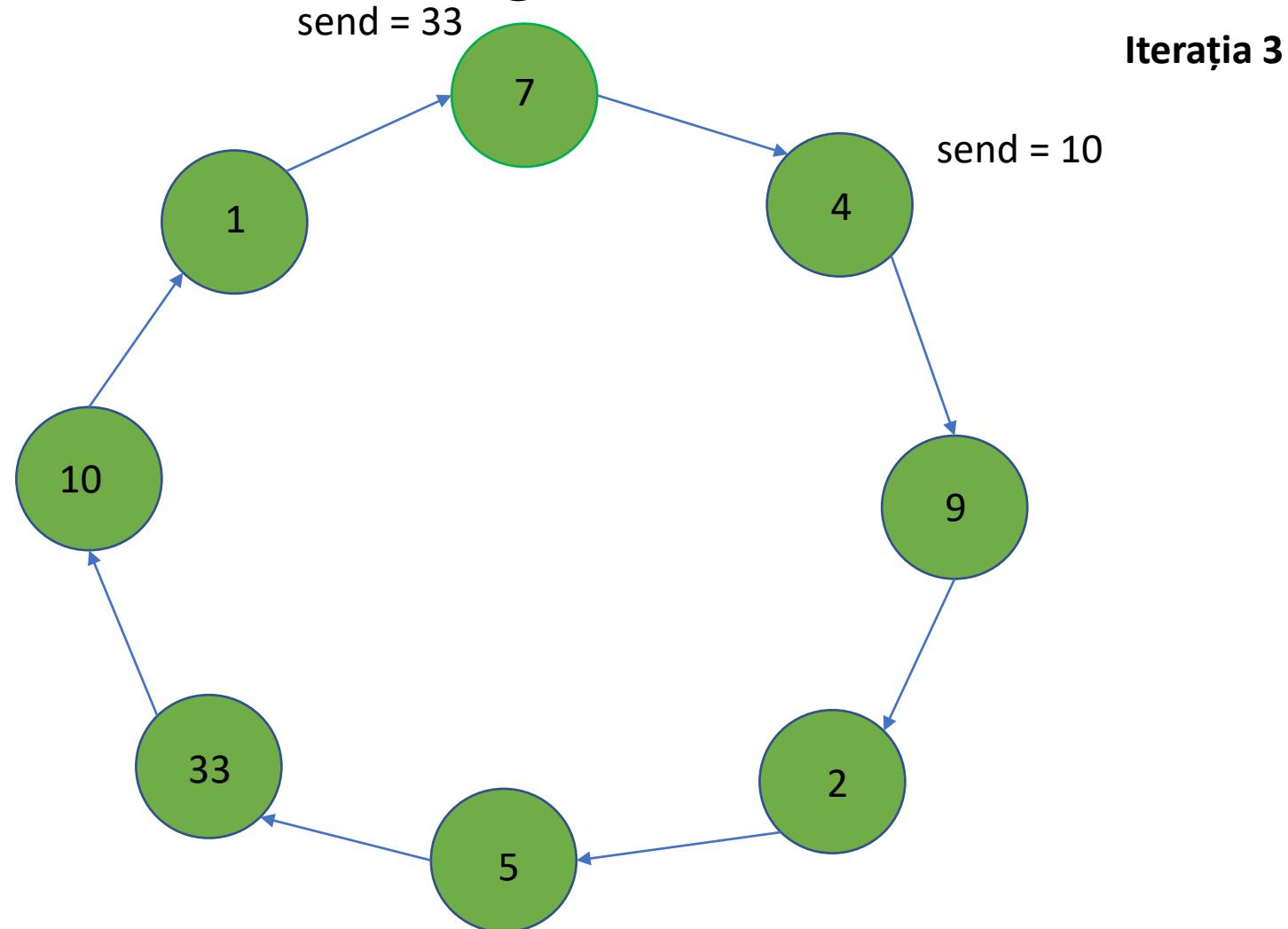


Algoritmul Flooding (LCR)

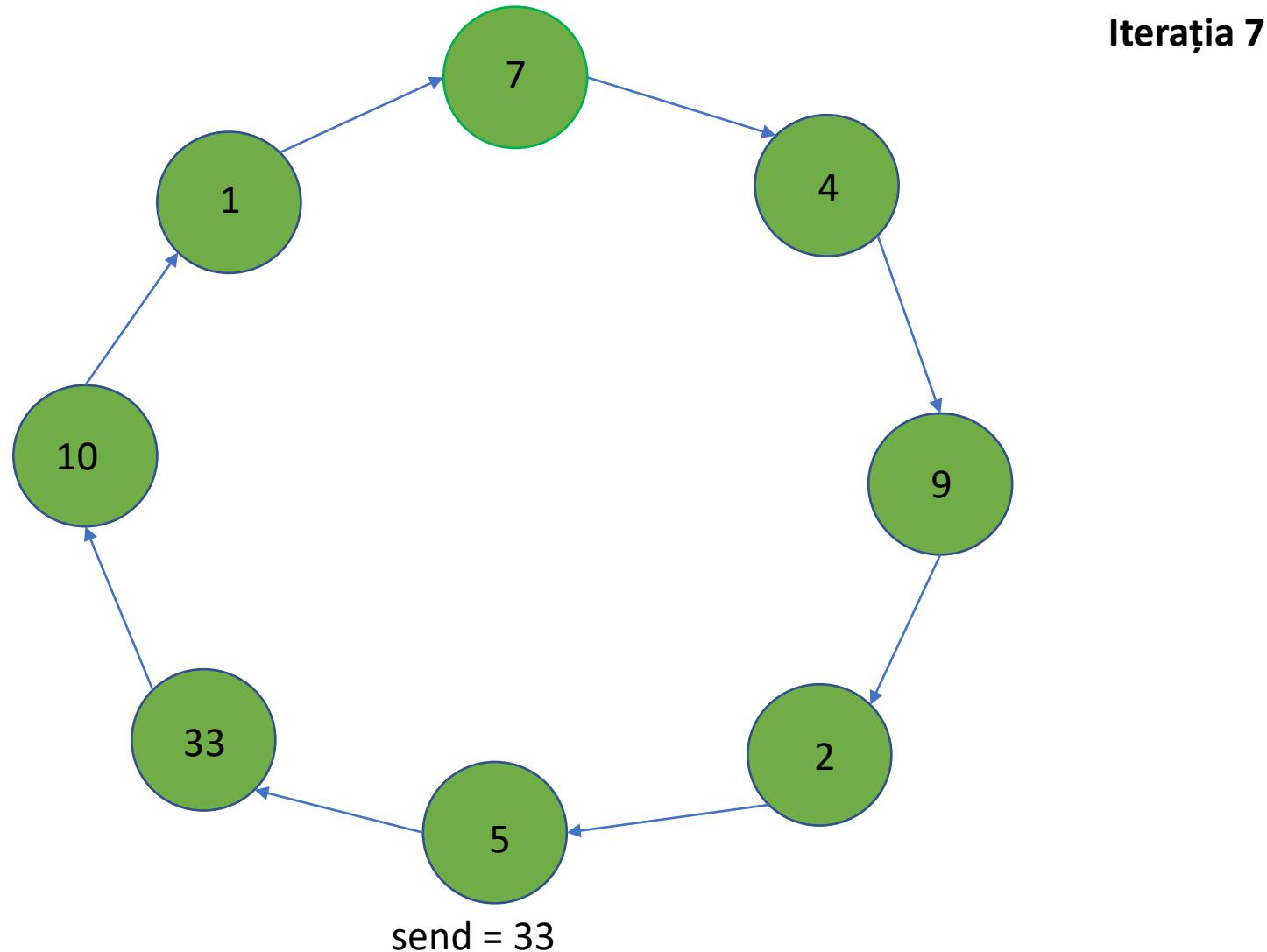
Iterația 3



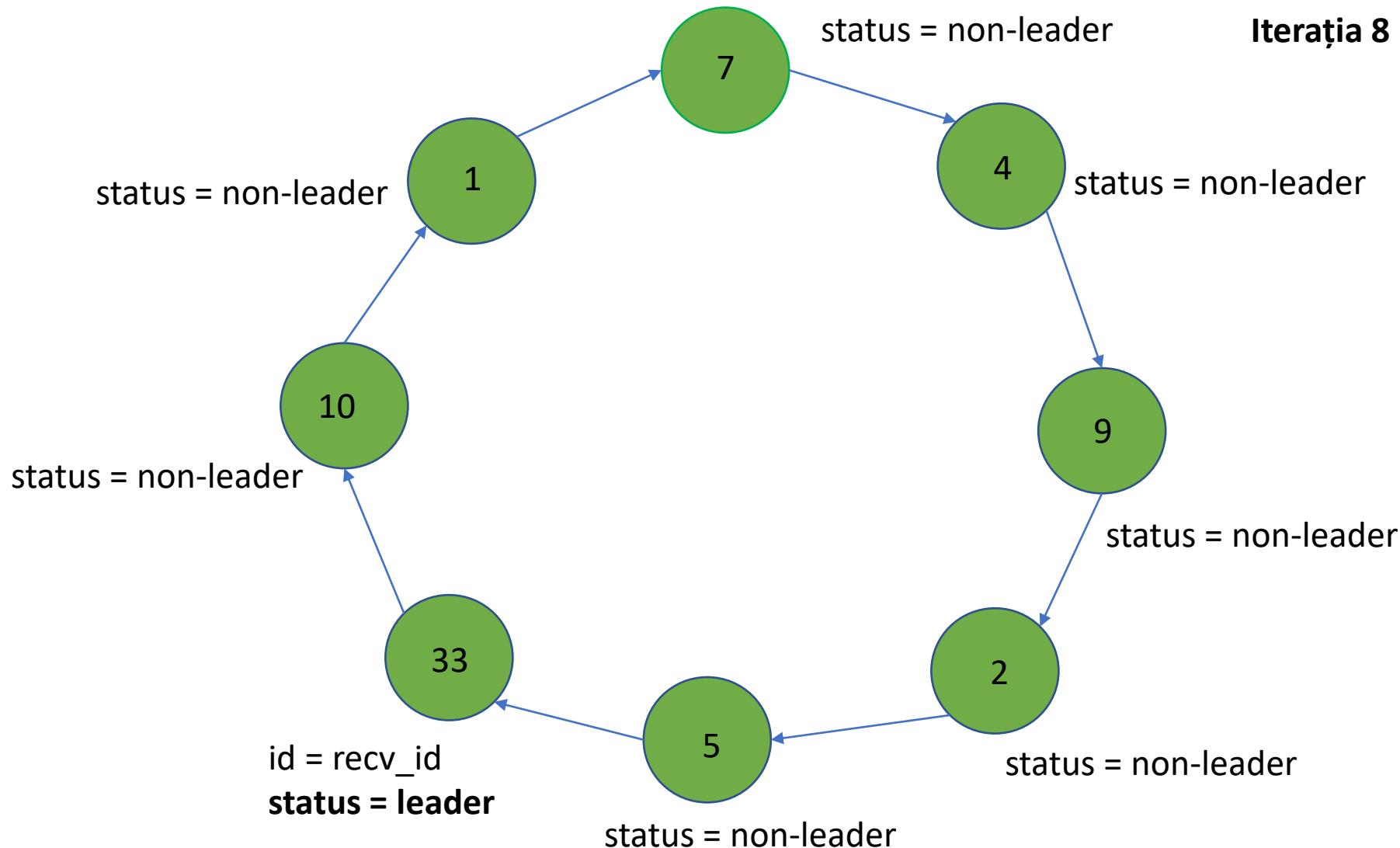
Algoritmul Flooding (LCR)



Algoritmul Flooding (LCR)



Algoritmul Flooding (LCR)



- În varianta curentă nu avem criteriu de oprire (nodurile rulează la infinit)
- **Solutie:** Pentru oprire liderul poate difuza un mesaj de raport, prin care semnalează încheierea competiției

Algoritmul Flooding (LCR)

Algoritm **Flooding()**:

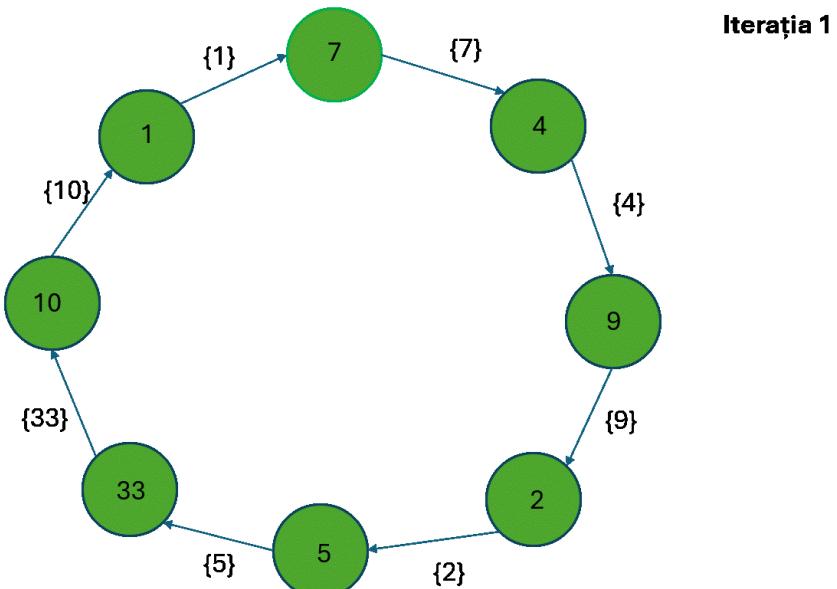
M_i : - int id (id propriu)
- int send_id (var auxiliară), inițial id
- status $\in \{\text{lider}, \text{non-lider}\}$, inițial non-lider

Funcție transformare nod i ():

1. send(send_id, index + 1 mod n);
2. recv_id = recv(index - 1 mod n);
3. **If** (recv_id > id):
 1. send_id := recv_id;
4. **ElseIf** (recv_id == id): status = leader;

Teorema [Lynch]. Algoritmul LCR rezolvă problema alegerii liderului.

Complexitate. Complexitatea timp este n iterări până la anunțarea unui lider, iar complexitatea mesaj este $O(n^2)$.



Algoritmul Flooding (topologie generală)

- Reconsiderăm problema AL, de data aceasta într-o topologie generală reprezentată printr-un graf directat **tare conectat**.
- Fiecare nod are un ID unic, ales dintr-un spațiu total ordonat.
- Un singur nod poate fi ales LIDER în rețea.
- Ideea unui algoritm simplu (extensie LCR):
 - Fiecare proces înregistrează ID-ul maxim cunoscut până în prezent
 - La fiecare iterare, nodurile propagă acest maxim pe toate muchiile de ieșire
 - După *diam* iterării, dacă ID-ul maxim coincide cu ID-ul propriu, nodul se va declara LIDER, altfel NON-LIDER.

Algoritmul Flooding (graf tare conectat)

Algoritm **Flooding_Gen**(max()):

- M_i : - int id (id propriu)
- int max_id (var auxiliară), inițial id
- $status \in \{\text{lider, non-lider}\}$, inițial non-lider
- int $rounds$, integer, inițial 0
- int $diam$ (diametru graf)

Funcție transformare nod i ():

1. $t := t + 1$
2. Fie U mulțimea ID-urilor primite de la vecinii de intrare
3. $max_id := \max(\{max_id\} \cup U)$
4. **If** ($rounds == diam$):
 1. **If** ($max_id == id$): $status = \text{leader}$;
 2. **Else**: $status = \text{non-leader}$;
5. **Else**: $\text{send}(max_id, \text{vecini ieșire})$

Teorema [Lynch]. În algoritmul Flooding, nodul cu indicele i_{max} este lider, restul nodurilor non-lider, după $diam$ iterații.

Complexitate. Complexitatea timp este $diam$ iterații până la anunțarea unui lider, iar complexitatea mesaj este $diam \cdot |E|$. Prin $|E|$ înțelegem numărul de muchii directate din graf.

Remarci.

1. Flooding reprezintă o generalizare a LCR;
2. LCR nu necesită informație globală;
3. Dacă graful = inel unidirecțional, atunci $diam \cdot |E| = (n - 1) \cdot n \approx n^2$;
4. Algoritmul funcționează cu o aproximare a constantei $diam$;

Algoritmul Flooding (alte aplicații)

- Difuzare în rețea (Broadcast)
- Arbori Breadth-First Search
- Consens (sincronizat)
- Estimari globale:
 - Diametru
 - Număr de noduri
 - Calcul distribuit

Sisteme și algoritmi distribuiți

Curs 4

Problemă și ipoteze

Problema diseminării de jetoane: fie un set de jetoane (*tokens*) $M = \{v_1, v_2, \dots, v_k\}$, $|M| = k$, distribuite pe nodurile din rețea. Nodul i stochează în starea inițială doar token-ul propriu $v_i \in M$. Realizați distribuția jetoanelor astfel încât, în starea optimă orice nod i va deține toate jetoanele din M .

- Topologie reprezentată printr-un graf directat **tare conectat**.
- Fiecare nod are un **ID unic** id_i .
- Dimensiune token $\text{sizeof}(v_i) = \text{sizeof}(ID) = B$ biți.
- Notații:
 - \mathcal{N}_i^- mulțimea vecinilor de intrare asociați nodului $i \in V$
 - \mathcal{N}_i^+ mulțimea vecinilor de ieșire asociați nodului $i \in V$
 - $< m_1, m_2 >$ atașarea mesajului m_2 la m_1

Algoritmul FloodSet

Algoritm **FloodSet()**:

M_i : - int id (id propriu)

- int v (token, inițial egal cu $x_i(0)$)
- int t , integer, inițial 0

Funcție transformare nod i ():

1. Fie U mulțimea mesajelor $\langle v_j, id_j \rangle$ primite de la \mathcal{N}_i^-
2. $M(t+1) = M(t) \cup U$
3. Fie $V(t+1)$ mulțimea valorilor v_j din $M(t+1)$
4. Fie $I(t+1)$ mulțimea id-urilor id_j din $M(t+1)$
5. **If** ($I(t+1) == I(t)$): **STOP**;
6. **Else**: send($M(t+1), \mathcal{N}_i^+$)
7. $t := t + 1$

Teorema [Kuhn et al.]. În algoritmul FloodSet, $M_i(t) = M$ după $O(diam)$ iterații.

1. FloodSet folosește mesaje $O(n B)$.
2. FloodSet necesită memorie $O(n B)$.
3. FloodSet nu necesită cunoașterea lui n sau $diam$.
4. Analiza complexității este similară cu cea din cazul **Flooding_gen(max())**.
5. FloodSet este un tipar algorithmic care se poate aplica pentru calcularea oricărei funcții.

Adaptare FloodSet pentru calcul distribuit

Aplicație: Calculați distribuit valoarea funcției f în x , i.e. $x_i^* = f(x_1, x_2, \dots, x_n)$.

- Considerăm $v_i := x_i$, i.e. $M = \{x_1, x_2, \dots, x_n\}$, $|M| = n$.
- Nodul i pornește din $x_i(0) := v_i$, $\forall i \in V$, converge către $x_i^* = f(x_1, \dots, x_n)$.

Păstrăm ipotezele anterioare:

- Topologie reprezentată printr-un graf directat **tare conectat**.
- Fiecare nod are un **ID unic** id_i .
- Dimensiune token $\text{sizeof}(v_i) = \text{sizeof}(ID) = B$ biți.

Adaptare FloodSet pentru alte funcții

Algoritm **FloodSet**($f, x(0)$):

- M_i : - int id (id propriu)
- int v (token, inițial egal cu x_i)
- funcție obiectiv $f()$
- int t , integer, inițial 0

Funcție transformare nod i ():

1. Fie U mulțimea mesajelor $\langle v_j, id_j \rangle$ primite de la \mathcal{N}_i^-
2. $M(t+1) = M(t) \cup U$
3. Fie $V(t+1)$ mulțimea valorilor v_j din $M(t+1)$
4. Fie $I(t+1)$ mulțimea id-urilor id_j din $M(t+1)$
5. **If** ($I(t+1) == I(t)$):
 1. **Return** $f(V(t))$
6. **Else**: send($M(t+1), \mathcal{N}_i^+$)
7. $t := t + 1$

Teorema [Kuhn et al.]. Algoritmul $\text{FloodSet}(f, x(0))$ returnează valoarea lui $f(x(0))$ după $O(diam)$ iterații.

1. FloodSet folosește mesaje $O(n B)$.
2. FloodSet necesită memorie $O(n B)$.
3. FloodSet nu necesită cunoașterea lui n sau $diam$;
4. Analiza complexității este similară cu cea din cazul **Flooding_gen(max())**.

Ipoteze și premise

- Topologie reprezentată printr-un graf directat **tare conectat**.
- ~~Fiecare nod are un **ID unic** id_t .~~ (Rețele anonime)
- Fiecare nod are un token $v_i \subset M$, un M mulțime total ordonată.
- Dimensiune token $\text{sizeof}(v_i) = \text{sizeof}(ID) = B$ biți.
- Urmărim algoritmi cu necesar de memorie/mesaj $< O(nB)$.
- Teorema de imposibilitate pentru rețele anonime.

Teorema de imposibilitate pentru rețele anonime

Ipoteze model distribuit:

- Noduri identice
- Rețea anonimă
- Determinism
- Memorie locală limitată (e.g. creștere slabă funcție de gradul nodului)
- Absența informației globale (P_i cunoaște doar vecinii de intrare)
- Topologie statică

Teorema de imposibilitate pentru rețele anonime

O funcție $f: R^n \rightarrow R^n$ este *independentă de ordine și multiplicitate* dacă valoarea ei este complet determinată de mulțimea valorilor care apar în vectorul $x \in R^n$ (indiferent de ordinea și numărul de apariții), i.e.

$$\exists g \text{ a.î. } f(x) = g(\{v : \exists i : v = x_i\}).$$

Exemple:

- $f(x) = \max(x_1, \dots, x_n)$
- $f(x) = \min(x_1, \dots, x_n)$
- Contraexemplu: $f(x) = \frac{1}{n} \sum_{i=1}^n x_i$

Teorema de imposibilitate pentru rețele anonime

Teorema de imposibilitate [Hendrickx&Tsitsiklis]. Dacă o funcție f este calculabilă de modelul distribuit specificat anterior, atunci f este *independentă de ordine și multiplicitate*.

Concluzii:

- Algoritmii pentru Alegere Lider (e.g. Flooding) nu necesită informație globală pentru a rezolva problema AL
- Pentru a calcula funcții *dependente de ordine sau multiplicitate*, trebuie eliminată cel puțin o ipoteză a modelului.

Recapitulare alg. Flooding(max())

Algoritm **Flooding_Gen(max())**:

- M_i : - int v (token, inițial egal cu $x(0)$)
- int max_v (var auxiliară), inițial v
- $status \in \{\text{lider, non-lider}\}$, inițial non-lider
- int t , integer, inițial 0
- int $diam$ (diametru graf)

Funcție transformare nod i ():

1. $t := t + 1$
2. Fie U mulțimea token-urilor primite de la \mathcal{N}_i^-
3. $max_v := \max(\{max_v\} \cup U)$
4. **If** ($t == diam$):
 1. **If** ($max_v == v$): $status = \text{leader}$;
 2. **Else**: $status = \text{non-leader}$;
5. **Else**: $\text{send}(v, \mathcal{N}_i^+)$

Teorema [Lynch]. În algoritmul Flooding, nodul cu indicele i_{max} este lider, restul nodurilor non-lider, după $diam$ iterații.

Complexitate. Complexitatea timp este $diam$ iterații până la anunțarea unui lider, iar complexitatea mesaj este $diam \cdot |E|$. Prin $|E|$ înțelegem numărul de muchii directate din graf.

Recapitulare alg. Flooding(max())

Algoritm **Flooding_Gen(max())**:

M_i : - int v (token, inițial egal cu $x(0)$)
- int max_v (var auxiliară), inițial v
- $status \in \{\text{lider, non-lider}\}$, inițial non-lider
- int t , integer, inițial 0
- int $diam$ (diametru graf)

Starea $x_i(t)$ este tokenul maxim max_v la momentul t .

$$\mathcal{N}_i = \{x_{i_1}(t), \dots, x_{i_{|N_i|}}(t)\}$$
$$x_i(t+1) = \max(x_i(t), x_{i_1}(t), \dots, x_{i_{|N_i|}}(t))$$

Funcție transformare nod i ():

1. $t := t + 1$
2. Fie U mulțimea token-urilor primite de la \mathcal{N}_i^-
3. $max_v := \max(\{max_v\} \cup U)$
4. **If** ($t == diam$):
 1. **If** ($max_v == v$): $status = \text{leader}$;
 2. **Else**: $status = \text{non-leader}$;
5. **Else**: $\text{send}(v, \mathcal{N}_i^+)$

1. Flooding(max()) folosește mesaje $O(B)$
2. Flooding(max()) necesită memorie $O(B)$
3. Flooding(max()) necesită cunoașterea unei estimări a lui $diam$ (pt criteriu de oprire)

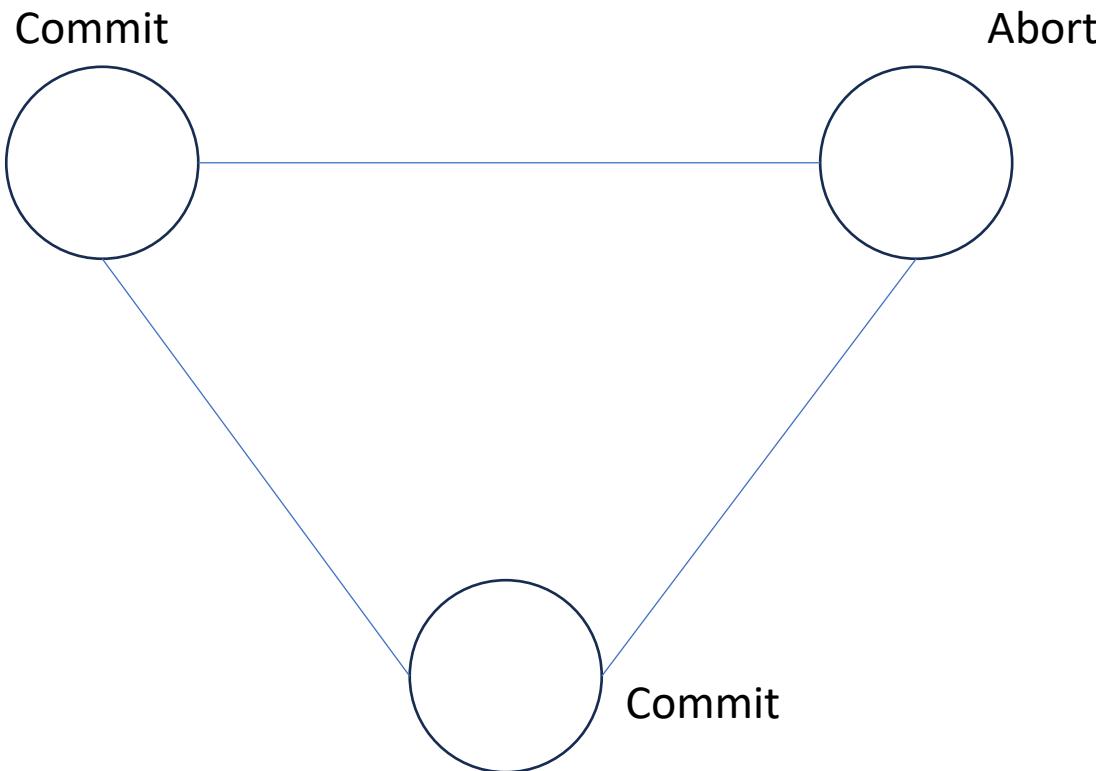
Problemă

Fie $a \in R^n$. Notăm $a_{[1]} \leq a_{[2]} \leq \dots \leq a_{[n]}$. Adaptați algoritmul Flooding pentru a calcula $a_{[n]} + a_{[n-1]}$.

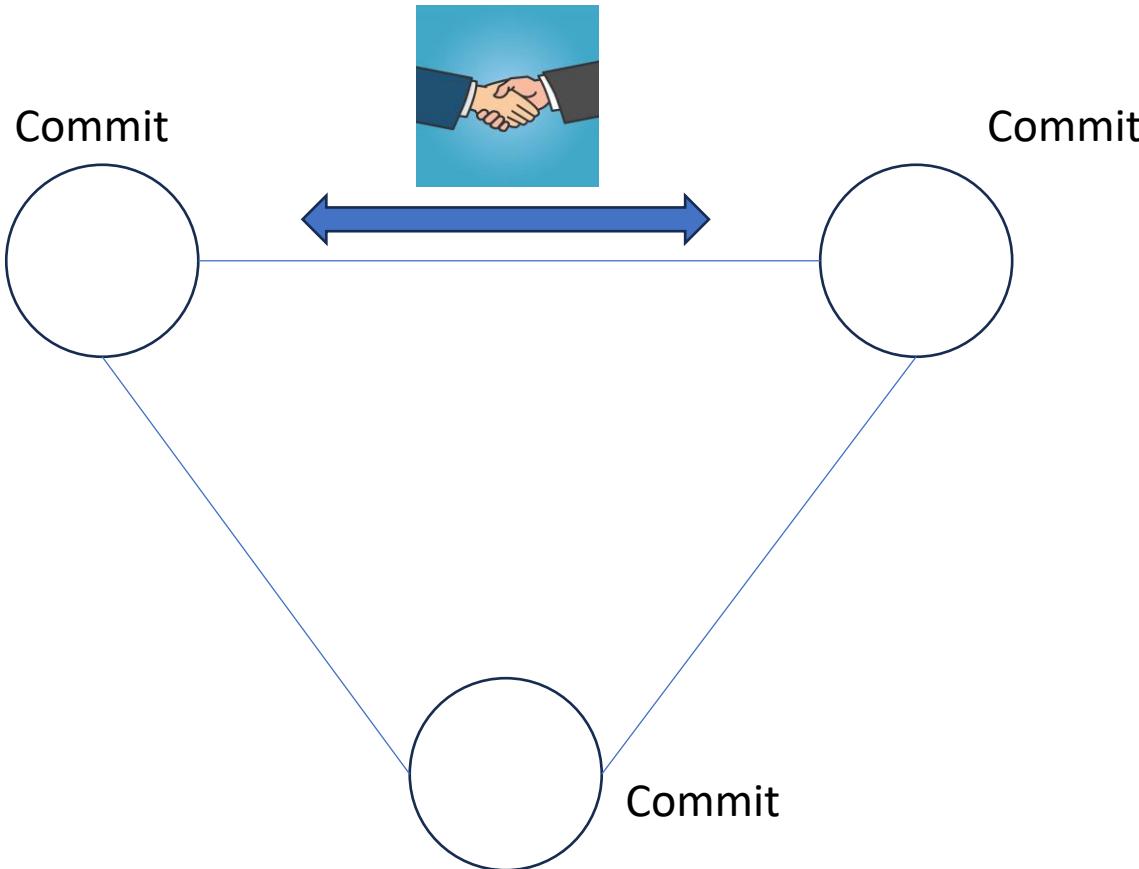
Rezolvare: la tablă.

Consens

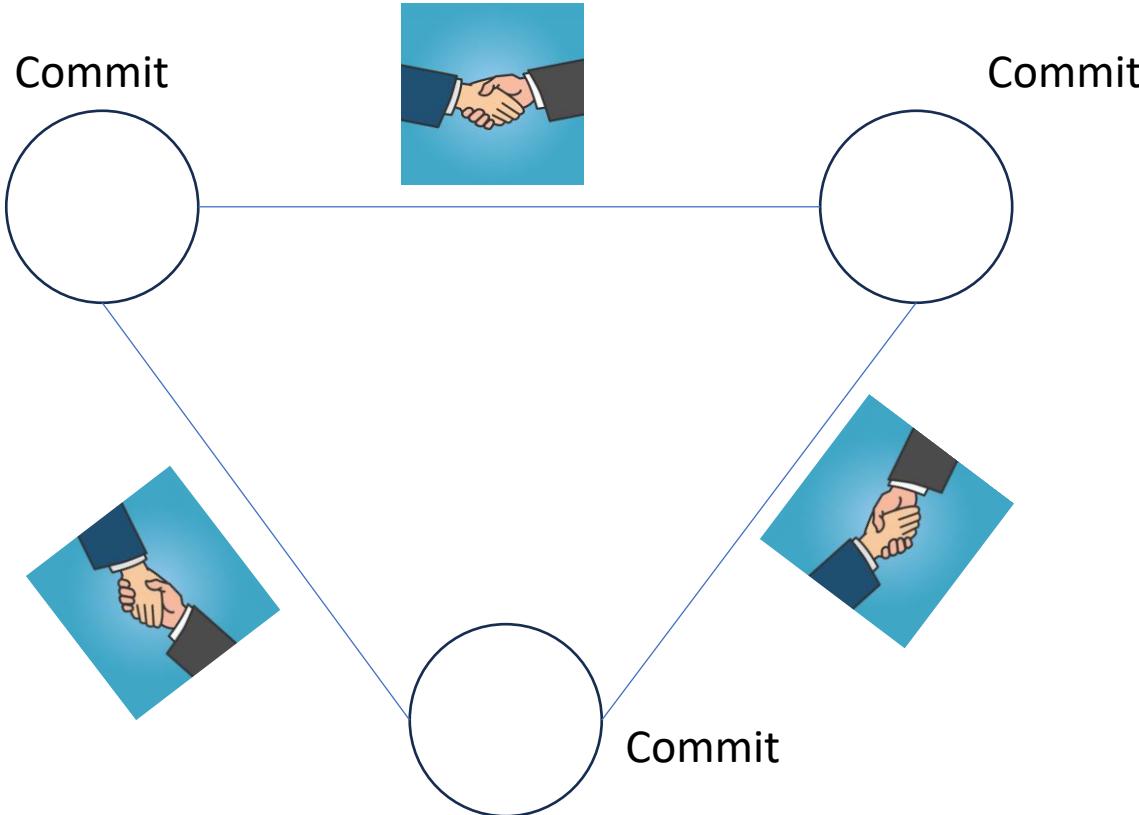
Consens



Consens

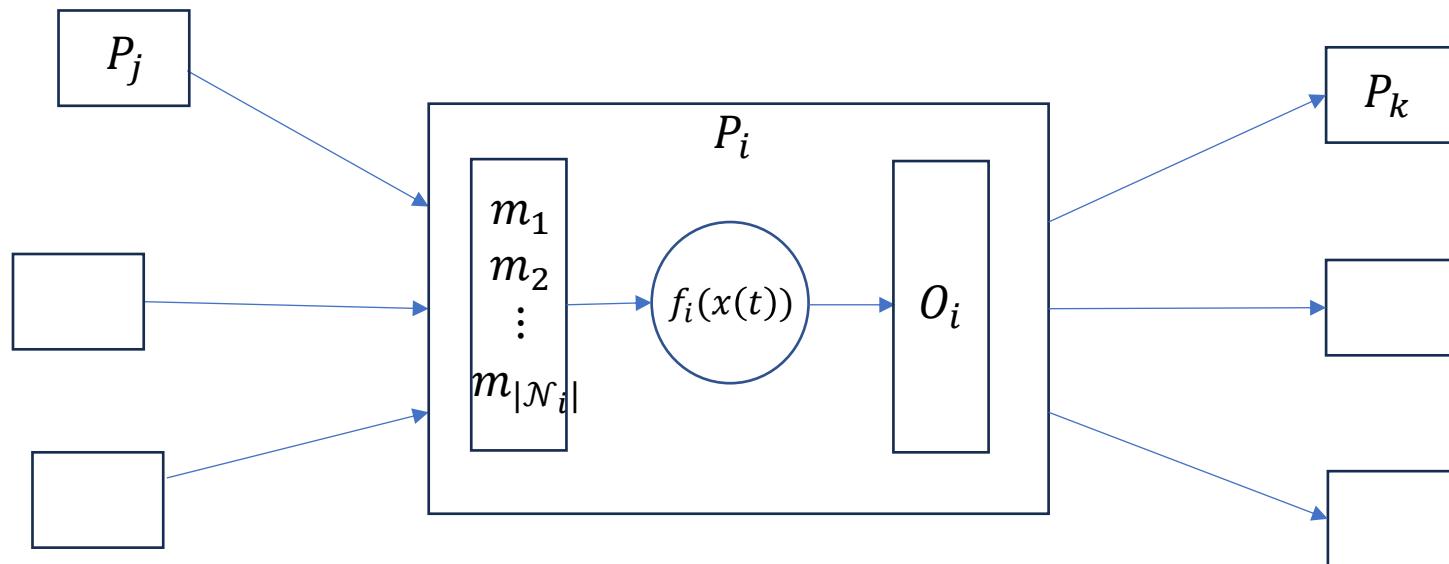


Consens



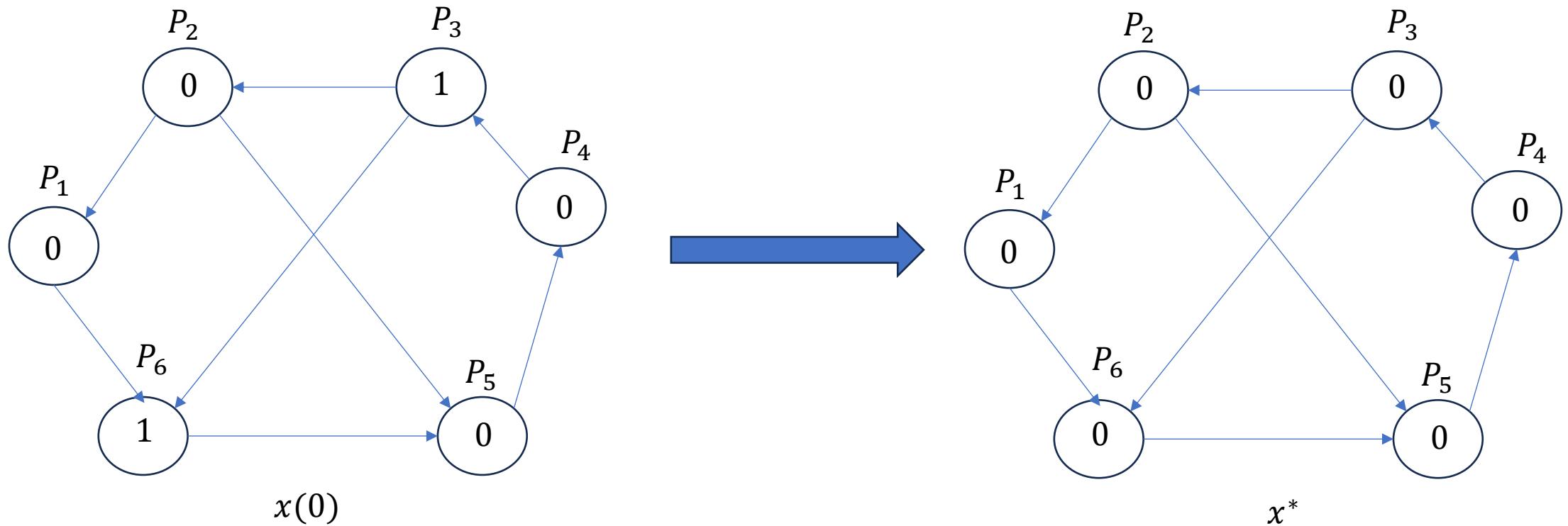
Consens: definiție

Starea (valoarea) uniformă a nodurilor unui sistem distribuit.



Consens: definiție

Starea (valoarea) uniformă a nodurilor unui sistem distribuit.



Consens: definiție

Starea (valoarea) uniformă a nodurilor unui sistem distribuit.

Condiția de acord: Nu există două procese 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 din sistem va decide eventual la un moment de timp.

Adesea se reduce la calcularea distribuită a valorii unei funcții de consens în starea inițială a sistemului.

Consens

Exemplu:

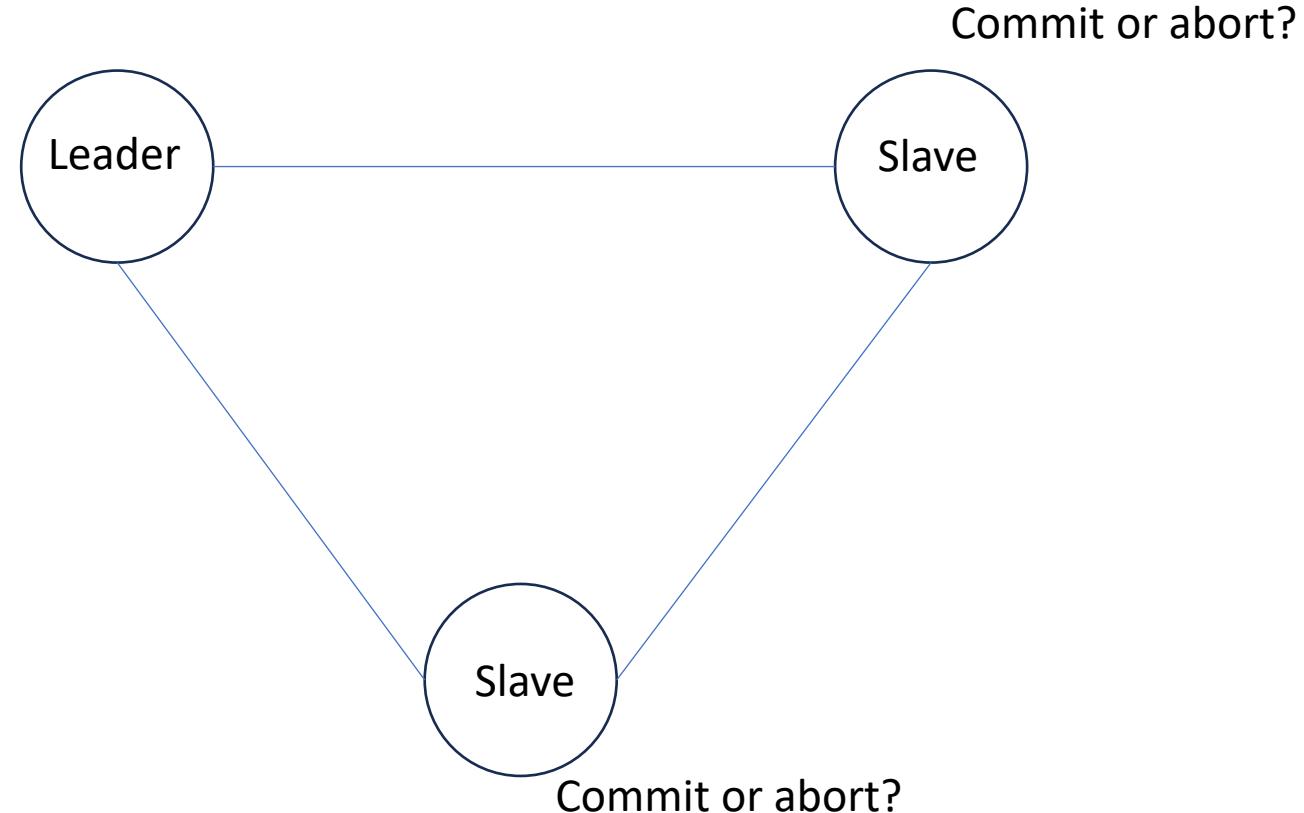
- Majoritar $x_i^* = Maj(x(0)) = \begin{cases} 1, & \text{dacă } |\{i|x_i(0) = 1\}| \geq \frac{n}{2} + 1 \\ 0, & \text{dacă } |\{i|x_i(0) = 1\}| < \frac{n}{2} + 1 \end{cases}$
 - Medie (aritmetică) $x_i^* = \frac{1}{n} \sum_{i=1}^n x_i(0)$
 - Mediană $x_i^* = x_{\left[\frac{n}{2}\right]}(0)$
 - Max-consens $x_i^* = \max_{1 \leq i \leq n} \{x_i(0)\}$
 - Min-consens $x_i^* = \min_{1 \leq i \leq n} \{x_i(0)\}$
- 
- Funcții independente de ordine și multiplicitate

Consens distribuit: aplicații

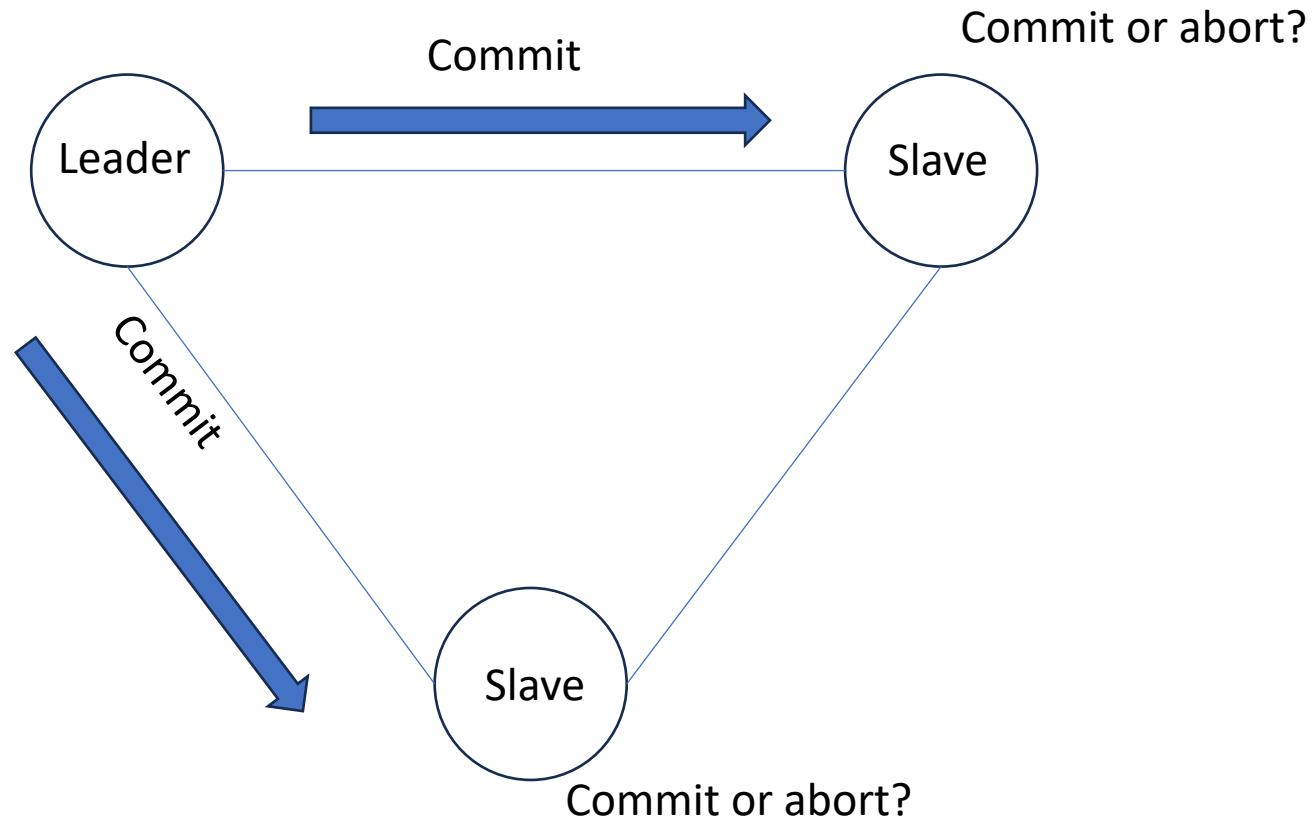
Calculul consensului este necesar în operații de nivel înalt:

- Electia liderului (max-consens)
- Sincronizare ceasuri (medie dinamică, max-consens dinamic)
- Asigurare consistență baze de date (majoritar)
- COMMIT distribuit (majoritar)
- Localizare distribuită în rețele de senzori (medie)
- Formarea de grupuri în rețele de agenți (medie dinamică)

Consens centralizat



Consens centralizat



Consens centralizat

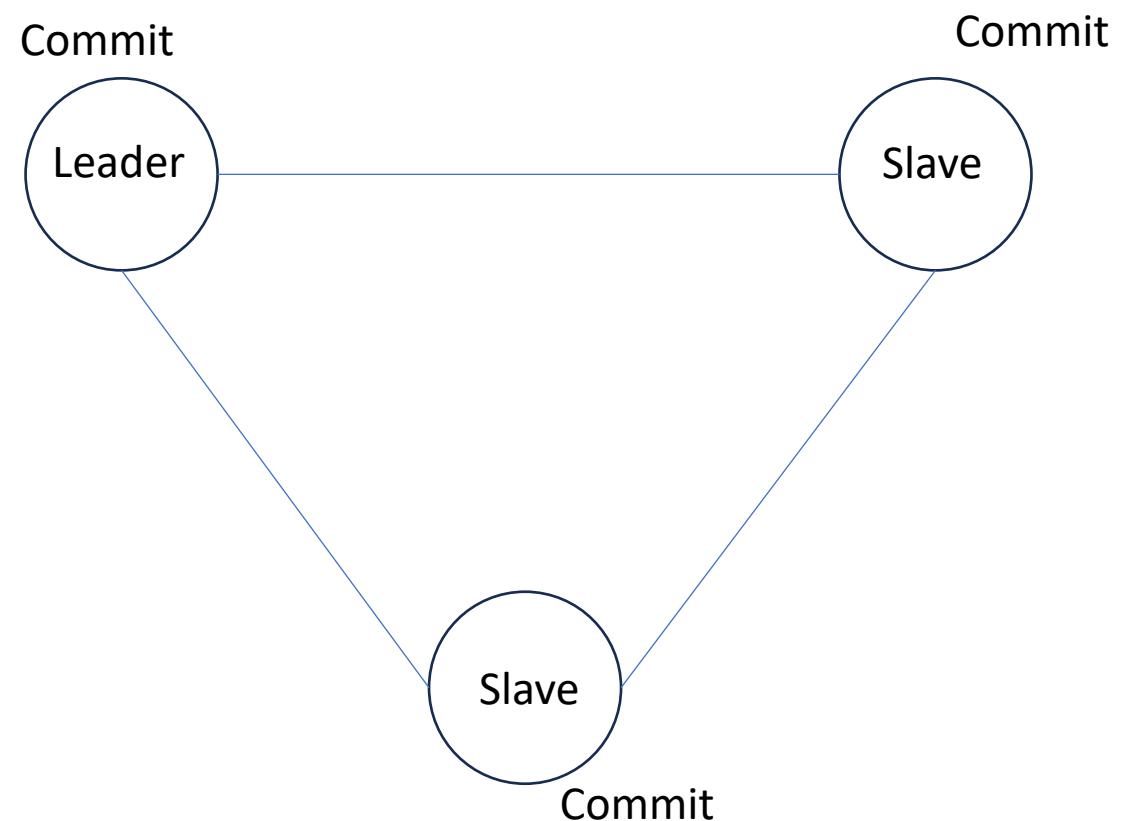
În context sincron fără defecte, asigurarea consensului centralizat se realizează printr-o simplă difuzarea de mesaje.

Dificultatea rămâne selecția preliminară a liderului, care se realizează folosind algoritmi sincroni AL (vezi cursul trecut).

Consens binar majoritar

- stare lider x_l

1. $\text{buf} = \text{Gather}(G);$
2. $x_l = \begin{cases} 1, & \text{dacă } |\{\text{buf}_i = 1\}| \geq \frac{n}{2} + 1 \\ 0, & \text{dacă } |\{\text{buf}_i = 1\}| < \frac{n}{2} + 1 \end{cases}$
3. $\text{Broadcast}(x_l);$



Consens centralizat

Consens binar majoritar (centralizat)

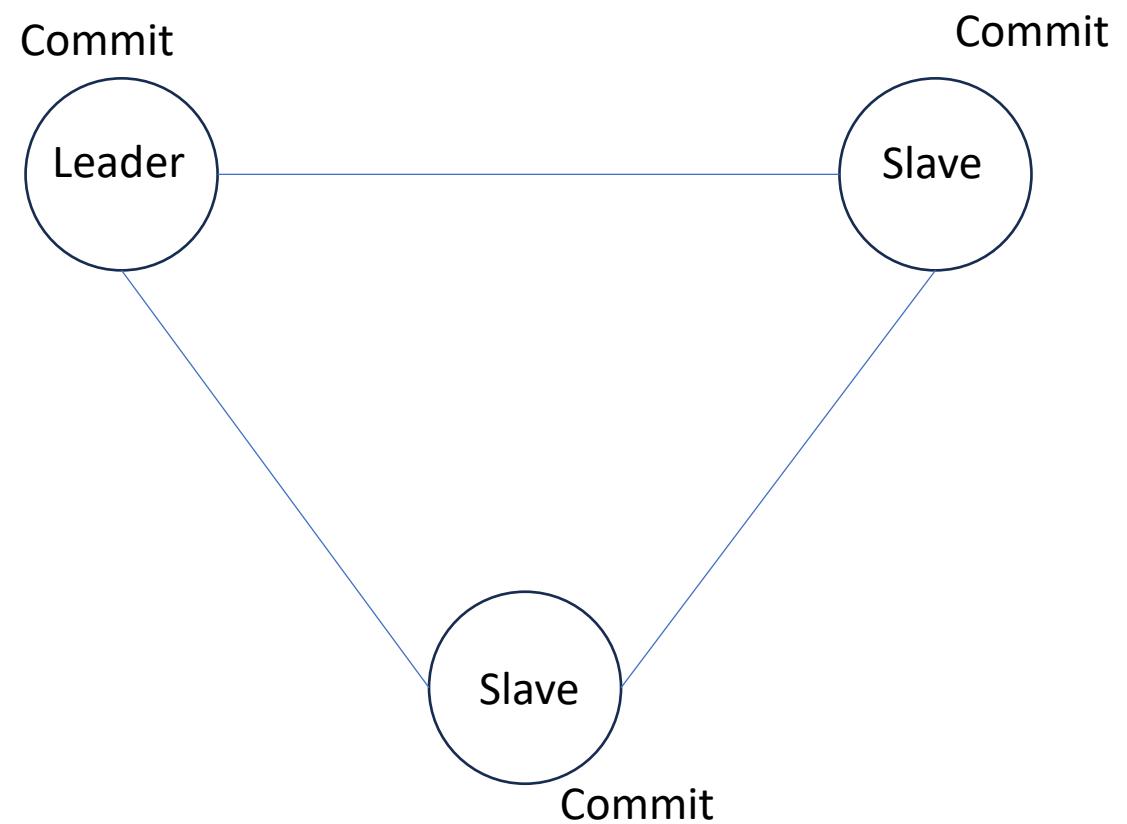
- stare lider $x_l \in \{0,1\}$

1. $\text{buf} = \text{Gather}(G);$

$$2. \quad x_l = \begin{cases} 1, & \text{dacă } |\{\text{buf}_i = 1\}| \geq \left\lceil \frac{n}{2} \right\rceil \\ 0, & \text{dacă } |\{\text{buf}_i = 1\}| < \left\lceil \frac{n}{2} \right\rceil \end{cases}$$

3. $\text{Broadcast}(x_l);$

- Slave: stochează 1 bit cu decizia curentă.
- Dacă avem perturbații pe noduri sau pe legături, schema de mai sus nu funcționează (rezultat posibil greșit).
- De asemenea, un lider defect impune reluarea procedurii de AL distribuit.



Consens distribuit

Cf. Teoremei de imposibilitate **Hendricx&Tsitsiklis**, dacă funcția de consens **nu este** independentă de ordine și multiplicitate atunci consensul este imposibil de atins fără cel puțin un atribut precum:

- informație globală *e.g.* $n, \text{diam}(G), G$
- capacitate locală de stocare mare $B > \deg(P_i)$
- o distribuție de identificatori

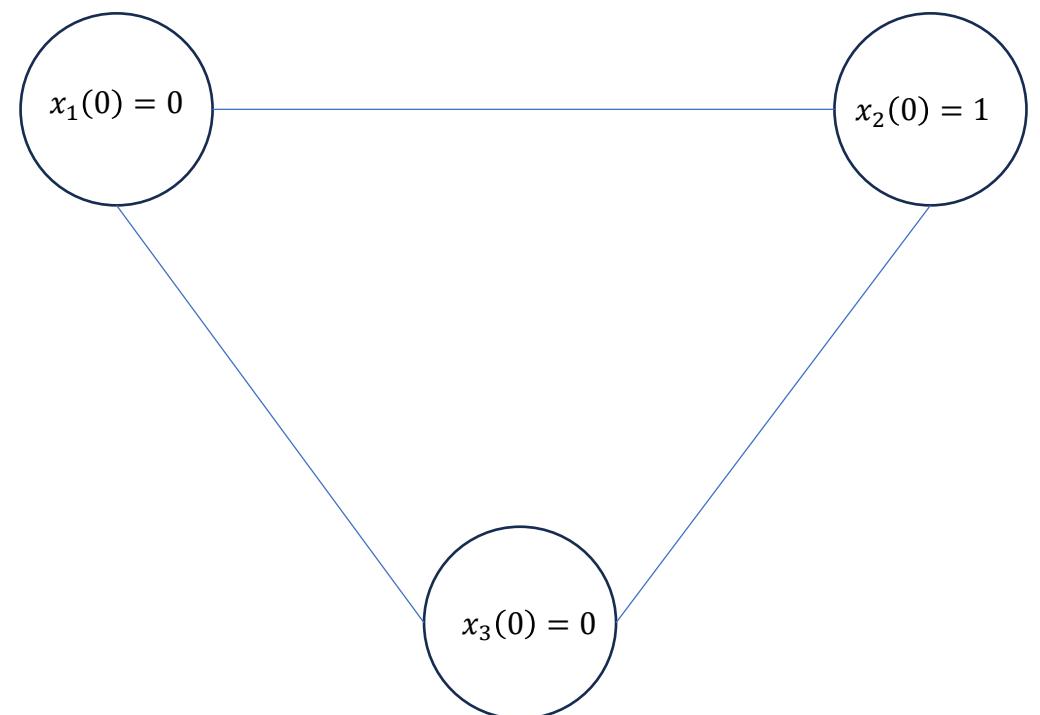
Consens binar majoritar (distribuit)

- stare lider $x_i \in \{0,1\}$

1. $\text{buf}_i = \text{Gather}(\mathcal{N}_i);$

$$2. \quad x_i = \begin{cases} 1, & \text{dacă } |\{\text{buf}_i[j] = 1\}| \geq \left\lceil \frac{|\mathcal{N}_i|}{2} \right\rceil \\ 0, & \text{dacă } |\{\text{buf}_i[j] = 1\}| < \left\lceil \frac{|\mathcal{N}_i|}{2} \right\rceil \end{cases}$$

3. $\text{Broadcast}(x_i, \mathcal{N}_i);$



Consens distribuit

Consens binar majoritar (distribuit)

- stare lider $x_i \in \{0,1\}$

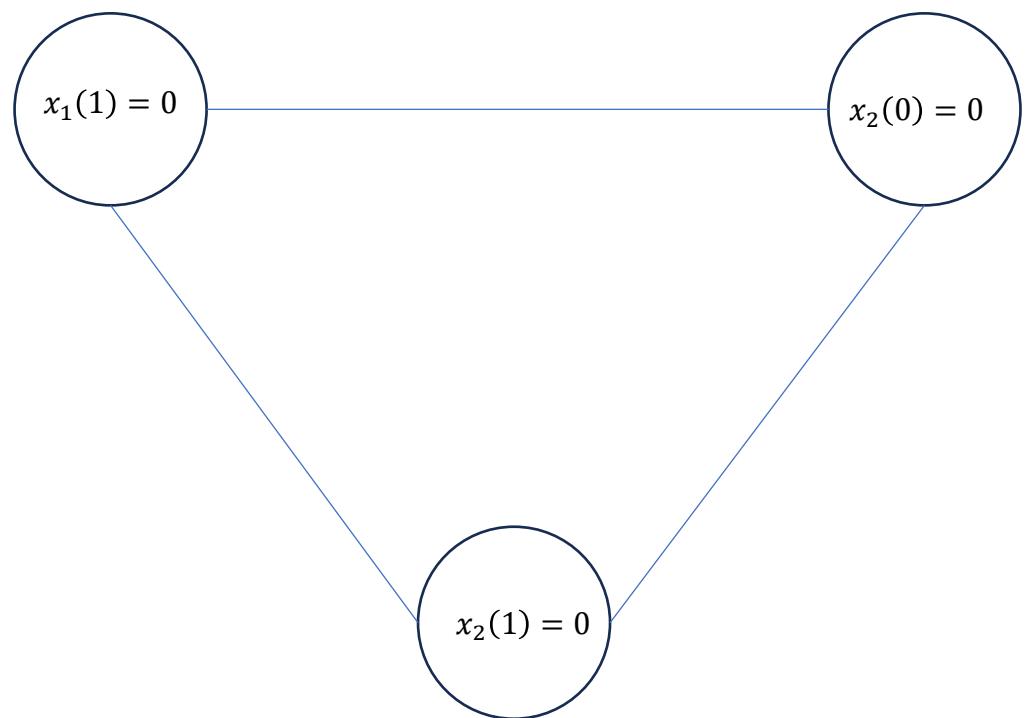
1. $buf_i = \text{Gather}(\mathcal{N}_i);$

2. $x_i = \begin{cases} 1, & \text{dacă } |\{buf_i[j] = 1\}| \geq \left\lceil \frac{|\mathcal{N}_i|}{2} \right\rceil \\ 0, & \text{dacă } |\{buf_i[j] = 1\}| < \left\lceil \frac{|\mathcal{N}_i|}{2} \right\rceil \end{cases}$

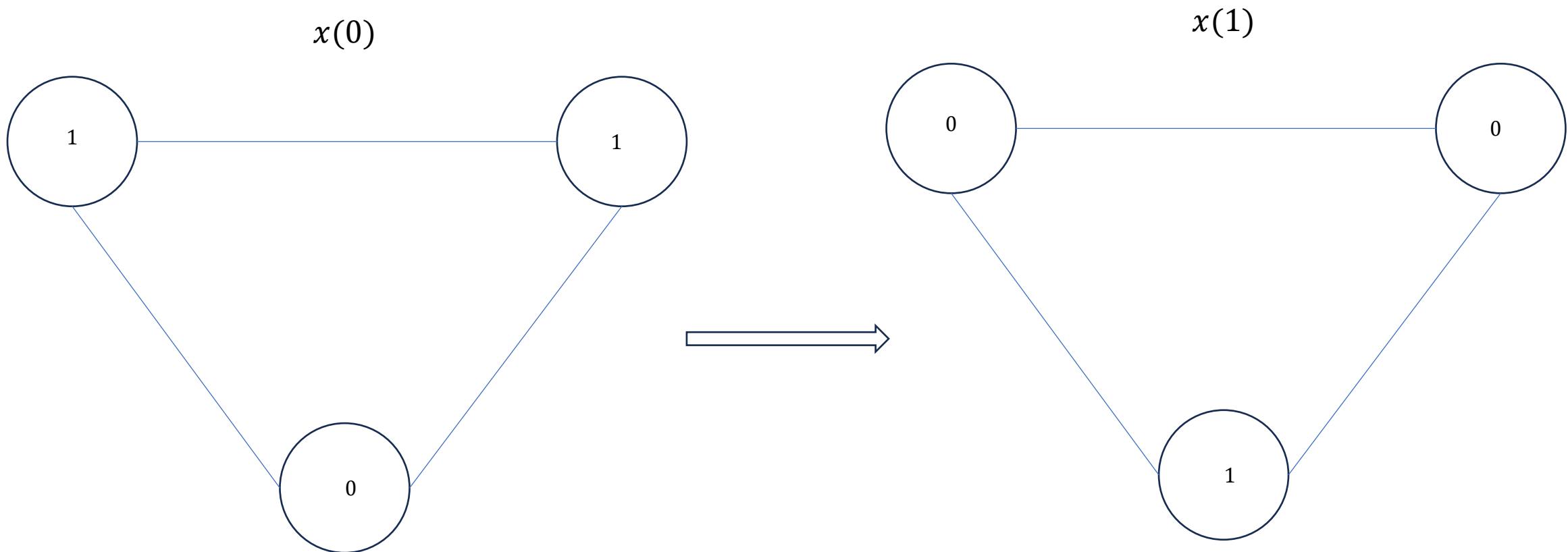
3. $\text{Broadcast}(x_i, \mathcal{N}_i);$

Actualizări locale bazate pe calculul „majorității” valorilor provenite de la vecini.

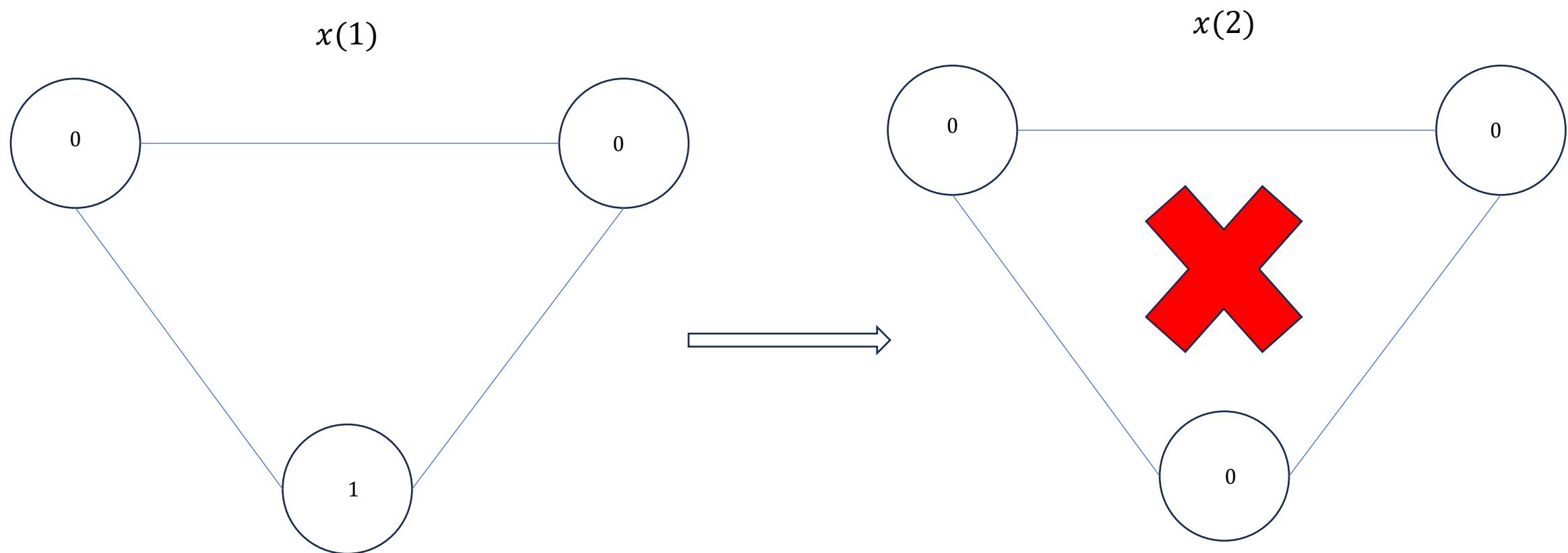
Dificultate majoră: natura binară a stărilor locale $x_i \in \{0,1\}$,
 $B = 1$.



Consens distribuit



Consens distribuit



Consens distribuit

Teoremă de imposibilitate [Land & Belew]. Fie sistemul $(\{x(t)\}_{t \geq 0}, \mathcal{G})$ cu n noduri și stări binare $x(t) \in \{0,1\}^n$. Nu există un algoritm determinist, sincron, distribuit care rezolvă exact *problema de consens binar majoritar* (pentru oricare \mathcal{G}).

Concluzie: Numărul (natura) stărilor per nod este un factor important în rezolvarea distribuită a problemelor centralizate.

LAND, Mark; BELEW, Richard K. No perfect two-state cellular automata for density classification exists. *Physical review letters*, 1995, 74.25: 5148.

Consens distribuit

Teoremă de imposibilitate [Land & Belew]. Fie sistemul $(\{x(t)\}_{t \geq 0}, \mathcal{G})$ cu n noduri și stări binare $x(t) \in \{0,1\}^n$. Nu există un algoritm determinist, sincron, distribuit care rezolvă exact *problema de consens binar majoritar* (pentru oricare \mathcal{G}).

Cum depăşim teorema de imposibilitate?

- Stări multiple (e.g. reale, nu binare), $B > 1$
- Automate probabilistice
- Automate asincrone

Algoritm FloodSet pentru consens

Păstrăm ipotezele anterioare:

- Topologie reprezentată printr-un graf directat **tare conectat**.
- Fiecare nod are un **ID unic** id_i .
- Dimensiune token $\text{sizeof}(v_i) = \text{sizeof}(ID) = B$ biți.

Problemă: Calculați distribuit valoarea funcției f de consens în $x(0)$, astfel încât $x_i^* = f(x(0))$.

- Considerăm $v_i := x_i$, i.e. $M = \{x_1, x_2, \dots, x_n\}$, $|M| = n$.
- Nodul i pornește din $x_i(0) := v_i$, $\forall i \in V$, converge către $x_i^* = f(x(0))$.
- Problema este o aplicație a soluției problemei de diseminare de jetoane.

Algoritm FloodSet pentru consens

Algoritm **FloodSet($f()$)**:

- M_i :
 - int id (id propriu)
 - int v (token, inițial egal cu x_i)
 - funcție obiectiv $f()$
 - int t , integer, inițial 0

Funcție transformare nod i ():

1. Fie U mulțimea mesajelor $\langle v_j, id_j \rangle$ primite de la \mathcal{N}_i^-
2. $M(t+1) = M(t) \cup U$
3. Fie $V(t+1)$ mulțimea valorilor v_j din $M(t+1)$
4. Fie $I(t+1)$ mulțimea id-urilor id_j din $M(t+1)$
5. **If** ($I(t+1) == I(t)$):
 1. **Return** $f(M(t))$
6. **Else**: send($M(t+1), \mathcal{N}_i^+$)
7. $t := t + 1$

- Reducem operația de consens static la calculul unei funcții de consens $f(x(0))$
- Dezavantaje:
 1. FloodSet folosește mesaje $O(n B)$
 2. FloodSet necesită memorie $O(n B)$
- În general urmărim ca dimensiunea mesajelor/memoriei să fie o funcție slab crescătoare de numărul de noduri (e.g. $\log(n), n^{\frac{1}{p}}$)
- Consens majoritar: considerarea de stări reale ne conduce la algoritmi eficienți.

Consens majoritar

Fie $v \in R^n$, atunci

$$Maj(v) = \begin{cases} 1, & \text{dacă } |\{i | v_i = 1\}| \geq \frac{n}{2} + 1 \\ 0, & \text{dacă } |\{i | v_i = 1\}| < \frac{n}{2} + 1 \end{cases}$$

se rescrie notând $\bar{v} = \frac{1}{n} \sum_{i=1}^n v_i$

$$Maj(v) = \begin{cases} 1, & \text{dacă } \bar{v} \geq \frac{1}{2} \\ 0, & \text{dacă } \bar{v} < \frac{1}{2} \end{cases}$$

Consens majoritar

Păstrăm ipotezele anterioare:

- Topologie reprezentată printr-un graf directat **tare conectat**.
- Dimensiune token $\text{sizeof}(v_i) = \text{sizeof}(ID) = B$ biți.

Problemă: Calculați distribuit valoarea funcției $f() = \text{Maj}()$ de consens în $x(0)$, astfel încât $x_i^* = f(x(0))$.

- Considerăm $v_i := x_i$, i.e. $M = \{x_1, x_2, \dots, x_n\}$, $|M| = n$.
- Nodul i pornește din $x_i(0) := v_i$, $\forall i \in V$, converge către $x_i^* = f(x(0))$.
- Observație: $\text{Maj}(v) = \frac{1}{2} \left(1 + \text{sgn} \left(\text{Mean}(v) - \frac{1}{2} \right) \right)$

Algoritm Flooding pentru consens majoritar

Algoritm **Flooding(Maj())**:

M_i : - int v (token)
- int d (grad intrare), integer
- int t , integer, inițial 0

- Inițial: $x_j(0) = v_j \in R$
- Iterație locală: $x_i(t + 1) = \frac{1}{d_i+1} \left(x_i(t) + \sum_{j \in \mathcal{N}_i^-} x_j(t) \right), \forall i$
- Analiza complexității timp pe scurt: la tablă!

Funcție transformare nod i ():

1. Fie \mathcal{U} mulțimea mesajelor $v_j = x_j(t)$ primite de la \mathcal{N}_i^-
2. $x(t + 1) = \frac{1}{d+1} \left(x_i(t) + \sum_{j \in \mathcal{N}_i^-} x_j(t) \right)$
3. **If** (criteriu_oprire):
 1. **Return** $\frac{1}{2} \left(1 + \text{sgn} \left(x(t) - \frac{1}{2} \right) \right)$
4. **Else**: send($x(t+1)$, \mathcal{N}_i^+)
5. $t := t + 1$

Sisteme și algoritmi distribuiți

Curs 5

Cuprins

- Consens – convergență (continuare curs 4)
- Defecte
- Algoritmi robusti la defect crash
- Algoritmi robusti la defect bizantin

Algoritm Flooding cu ponderi uniforme

Algoritm **Flooding**(Maj, v):

Mem_i: - int x_i (token), inițial v_i
- int d (grad intrare), integer
- int t , integer, inițial 0

Funcție transformare nod i ():

1. Fie U multimea mesajelor $v_j = x_j(t)$ primite de la \mathcal{N}_i^-
2. $x_i(t + 1) = \frac{1}{d+1} \left(x_i(t) + \sum_{j \in \mathcal{N}_i^-} x_j(t) \right)$
3. **If** (criteriu_oprire):
 1. **Return** $\frac{1}{2} \left(1 + \text{sgn} \left(x(t) - \frac{1}{2} \right) \right)$
4. **Else**: send($x(t+1)$, \mathcal{N}_i^+)
5. $t := t + 1$

Algoritm Flooding pentru consens

- Inițial: $x_j(0) = v_j \in R$
- Iterație locală:

$$x_i(t+1) = \frac{1}{d_i + 1} \left(x_i(t) + \sum_{j \in \mathcal{N}_i^-} x_j(t) \right), \quad \forall i$$

Mai pe larg: actualizarea lui $x_i(t+1)$ se face pe baza mediei aritmetice dintre starea $x_i(t)$ și stările vecinilor $x_j(t), j \in \mathcal{N}_i^-$; presupunem un transfer cu succes al stărilor $x_j(t)$ către P_i . Vectorial avem:

$$\begin{bmatrix} x_1(t+1) \\ \dots \\ x_n(t+1) \end{bmatrix} = \begin{bmatrix} \frac{1}{d_1 + 1} \left(x_1(t) + \sum_{j \in \mathcal{N}_1^-} x_j(t) \right) \\ \dots \\ \frac{1}{d_n + 1} \left(x_n(t) + \sum_{j \in \mathcal{N}_n^-} x_j(t) \right) \end{bmatrix} = \begin{bmatrix} \frac{1}{d_1 + 1} x_1(t) + \frac{1}{d_1 + 1} \sum_{j \in \mathcal{N}_1^-} x_j(t) \\ \dots \\ \frac{1}{d_n + 1} x_n(t) + \frac{1}{d_n + 1} \sum_{j \in \mathcal{N}_n^-} x_j(t) \end{bmatrix}$$

Observăm pe fiecare componentă a vectorului din partea dreaptă un produs scalar între stările nodurilor $\{i \cup \mathcal{N}_i^-\}$ și vectorul unidimensional $\tilde{a}_i = \frac{1}{d_i+1} [1 \ 1 \ \dots \ 1]^T$. Sau, echivalent, între vectorul coloană definit de

$$[a_i]_k = \begin{cases} \frac{1}{d_i+1}, & k \in \{i \cup \mathcal{N}_i^-\} \\ 0, & k \notin \{i \cup \mathcal{N}_i^-\} \end{cases} \text{ și vectorul stărilor } x(t).$$

Algoritm Flooding pentru consens

Algoritm Flooding de medie prezentat anterior se exprimă recurrent prin actualizarea liniară:

$$x_i(t+1) = a_{ii}x_i(t) + \sum_{j \in \mathcal{N}_i^-} a_{ij}x_j(t) \quad \forall i$$

Deci $x_i(t+1) = a_i^T x(t)$, iar dinamica stărilor sistemului este:

$$x(t+1) = Ax(t),$$

unde

$$A = \begin{bmatrix} a_1^T \\ \dots \\ a_n^T \end{bmatrix} \in R^{n \times n}, x(t) = \begin{bmatrix} x_1(t) \\ \dots \\ x_n(t) \end{bmatrix} \in R^n.$$

Matricea A este în strânsă legătură cu matricea de adiacență a grafului care determină topologia.

Care este structura matricii A pentru ponderi uniforme?

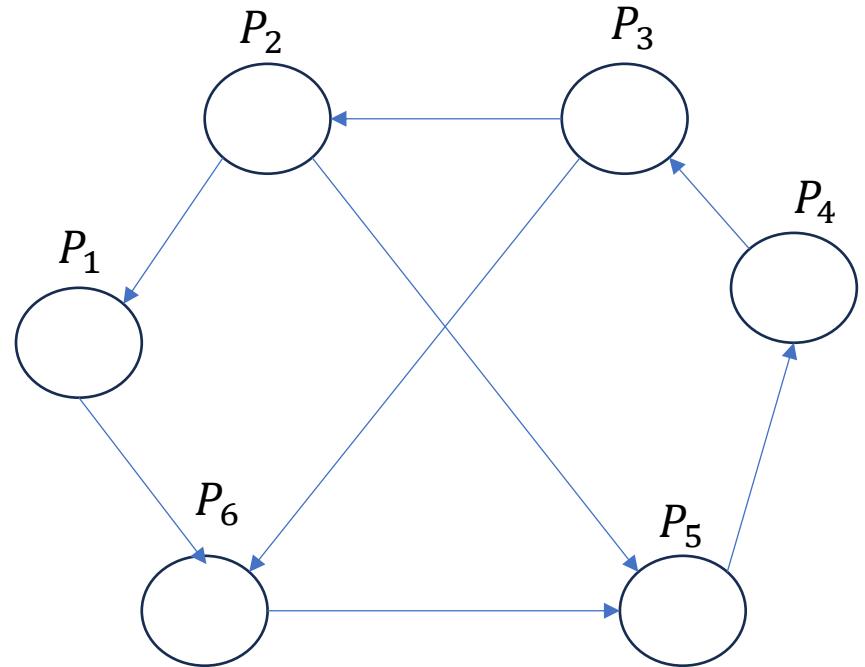
Exemple

Pentru graful alăturat matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 1/3 & 0 & 0 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 0 & 0 & 1/3 \end{bmatrix}$$

Definiție. Matricea A se numește *stochastică pe linii* dacă elementele $a_{ij} \geq 0$ și suma fiecărei linii este 1.

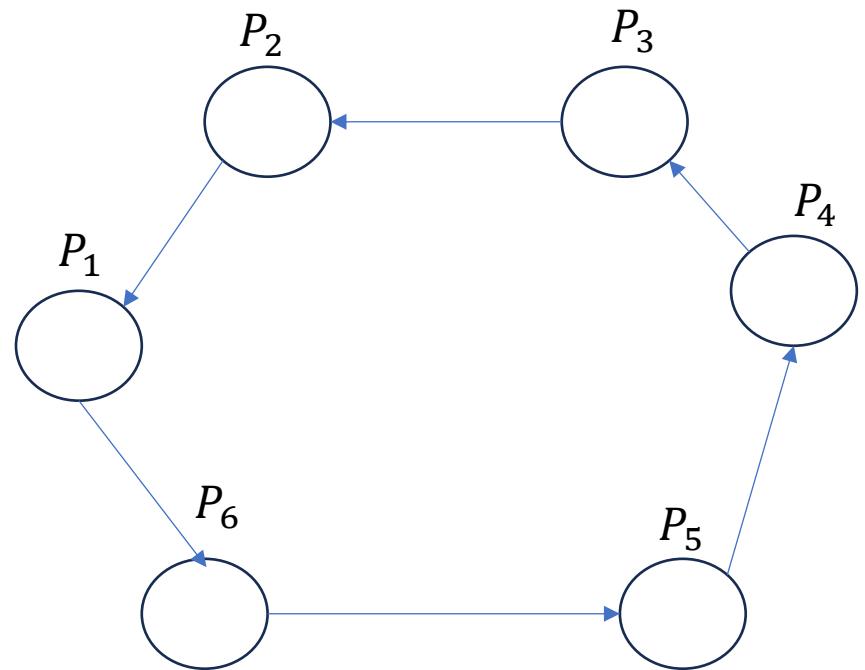
- Similar definim matricea stochastică pe coloane
- Matricile stochasticice pe linii și coloane se numesc *dublu stochastice*.
- Exemplul de mai sus unde se încadrează?
- Alte exemple? Cum generăm matrici dublu stochasticice?



Exemplu

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$

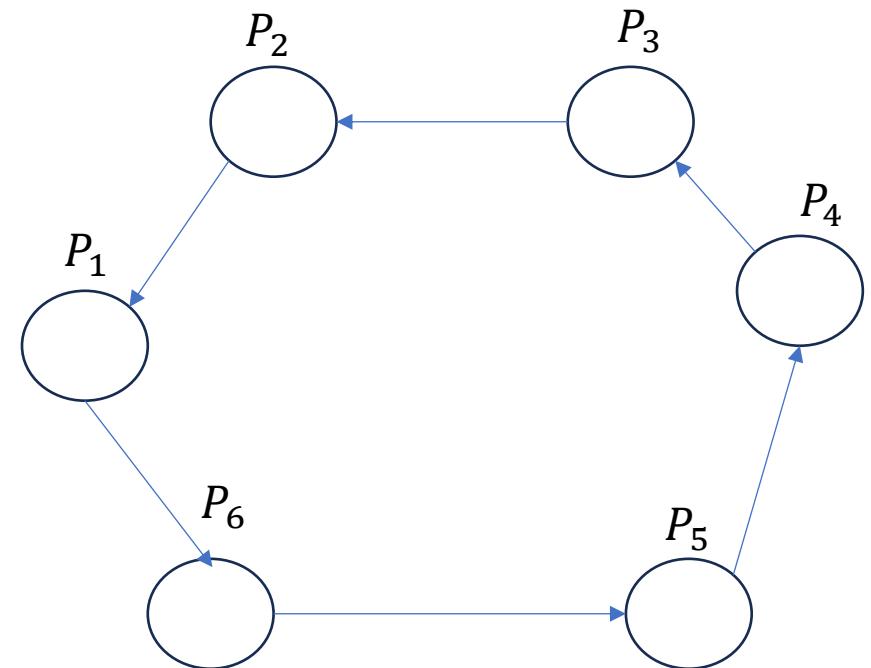


Exemple

În general, pentru un inel de dimensiune n , matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & \cdots & 0 & 0 \\ 0 & 1/2 & 1/2 & \ddots & 0 & 0 \\ \vdots & & & \ddots & & \vdots \\ 1/2 & 0 & 0 & \cdots & 0 & 1/2 \end{bmatrix} \in R^{n \times n}$$

- Dublu stochastică
- Nesimetrică (graf directat)

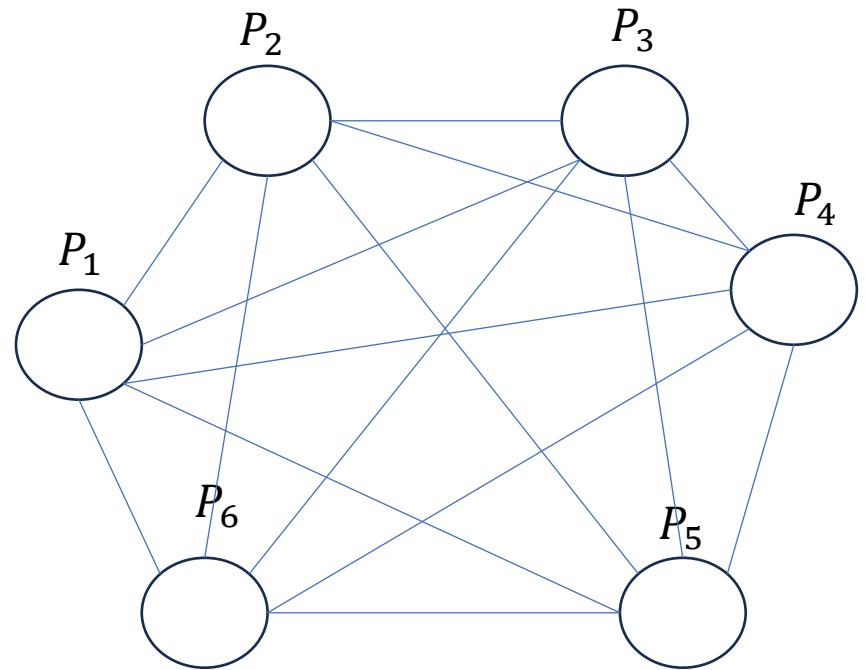


Exemplu

Pentru un graf de tip clică matricea asociată este:

$$A = \begin{bmatrix} 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \end{bmatrix}$$

- Dublu stochastică
- Simetrică



Algoritm Flooding pentru consens

Din dinamică stărilor

$$x(t+1) = Ax(t),$$

se observă ușor:

$$x(t) = A^t x(0),$$

de aceea convergența depinde total de comportamentul matricii A^t (implicit, doar de structura grafului).

Teorema. Dacă matricea A este stoastică pe linii, atunci se atinge consensul asymptotic, i.e. $x(t) \rightarrow c\mathbf{1}$ când $t \rightarrow \infty$. În plus, dacă matricea A este stoastică pe coloane (graful are grade de intrare uniforme), i.e. $\mathbf{1}^T A = A^T$, atunci

$$c = \frac{1}{n} \sum_{i=1}^n x_i(0).$$

- Rezultat valabil nu doar pentru ponderi uniforme.
- Condiția necesară pentru consens este ca matricea ponderilor să fie stoastică pe linii (fiecare să realizeze la fiecare iterație o combinație convexă între starea proprie și stările vecinilor)
- Dacă matricea ponderilor este, în plus, stoastică pe coloane, atunci valoarea de consens este media aritmetică a stărilor inițiale.

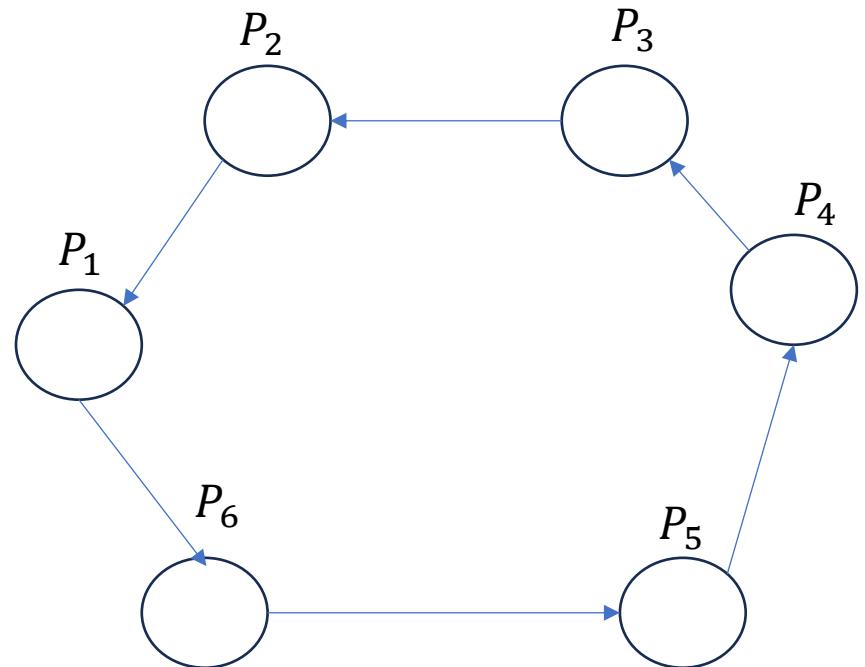
Exemplu

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$

A^2 :

$$\begin{array}{ccccccc} 0.2500 & 0.5000 & 0.2500 & 0 & 0 & 0 \\ 0 & 0.2500 & 0.5000 & 0.2500 & 0 & 0 \\ 0 & 0 & 0.2500 & 0.5000 & 0.2500 & 0 \\ 0 & 0 & 0 & 0.2500 & 0.5000 & 0.2500 \\ 0.2500 & 0 & 0 & 0 & 0.2500 & 0.5000 \\ 0.5000 & 0.2500 & 0 & 0 & 0 & 0.2500 \end{array}$$



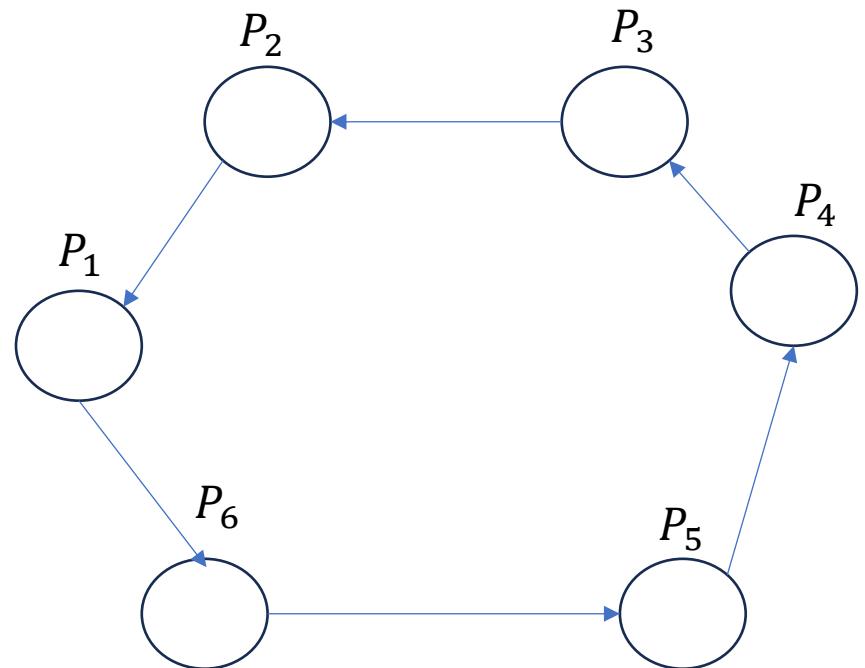
Exemplu

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$

A^4 :

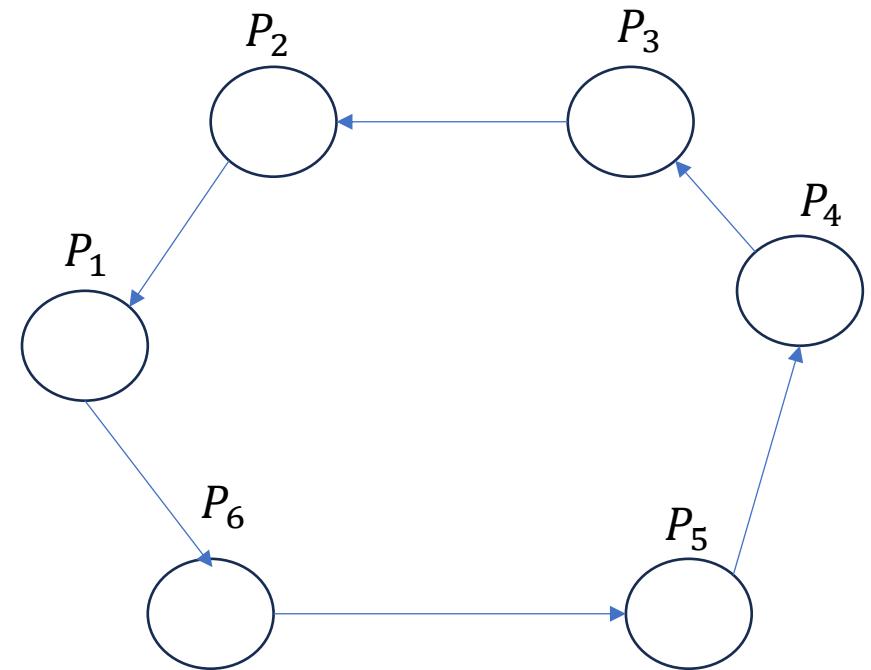
$$\begin{array}{cccccc} 0.0625 & 0.2500 & 0.3750 & 0.2500 & 0.0625 & 0 \\ 0 & 0.0625 & 0.2500 & 0.3750 & 0.2500 & 0.0625 \\ 0.0625 & 0 & 0.0625 & 0.2500 & 0.3750 & 0.2500 \\ 0.2500 & 0.0625 & 0 & 0.0625 & 0.2500 & 0.3750 \\ 0.3750 & 0.2500 & 0.0625 & 0 & 0.0625 & 0.2500 \\ 0.2500 & 0.3750 & 0.2500 & 0.0625 & 0 & 0.0625 \end{array}$$



Exemplu

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$



A^{10} :

0.206055	0.126953	0.087891	0.126953	0.206055	0.246094
0.246094	0.206055	0.126953	0.087891	0.126953	0.206055
0.206055	0.246094	0.206055	0.126953	0.087891	0.126953
0.126953	0.206055	0.246094	0.206055	0.126953	0.087891
0.087891	0.126953	0.206055	0.246094	0.206055	0.126953
0.126953	0.087891	0.126953	0.206055	0.246094	0.206055

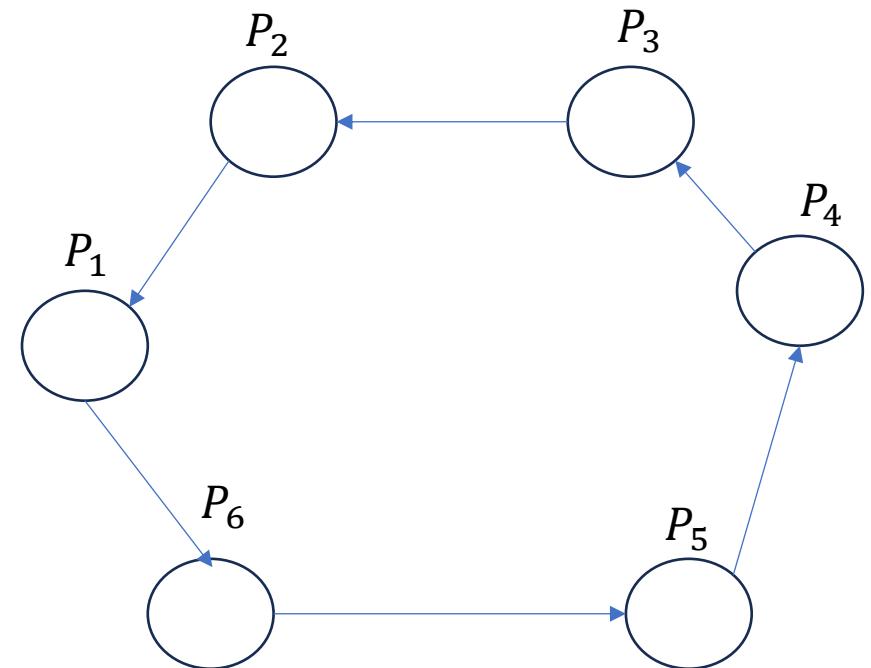
Exemplu

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$

A^{25} :

$$\begin{array}{ccccccc} 0.1746 & 0.1746 & 0.1667 & 0.1587 & 0.1587 & 0.1667 \\ 0.1667 & 0.1746 & 0.1746 & 0.1667 & 0.1587 & 0.1587 \\ 0.1587 & 0.1667 & 0.1746 & 0.1746 & 0.1667 & 0.1587 \\ 0.1587 & 0.1587 & 0.1667 & 0.1746 & 0.1746 & 0.1667 \\ 0.1667 & 0.1587 & 0.1587 & 0.1667 & 0.1746 & 0.1746 \\ 0.1746 & 0.1667 & 0.1587 & 0.1587 & 0.1667 & 0.1746 \end{array}$$



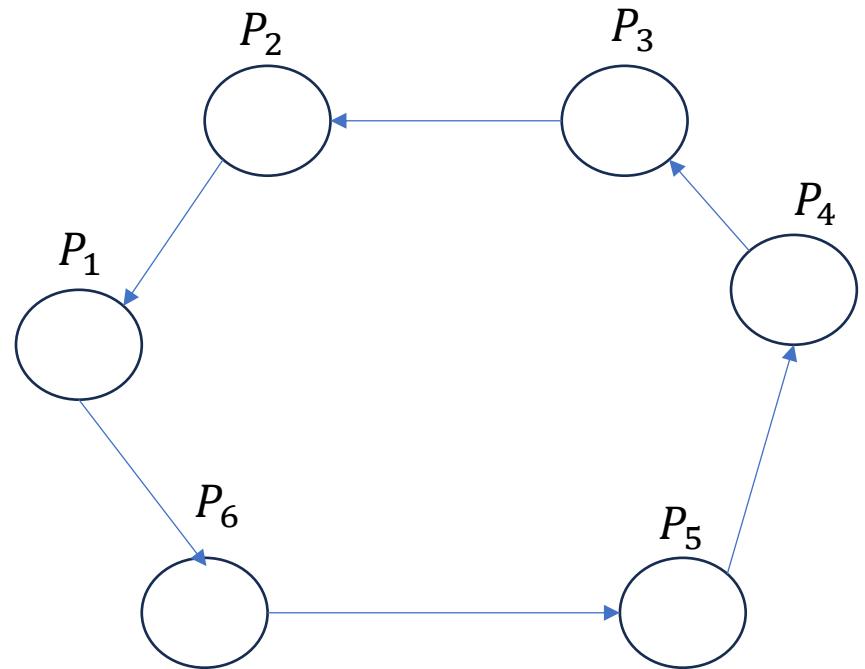
Exemplu

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$

A^{70} :

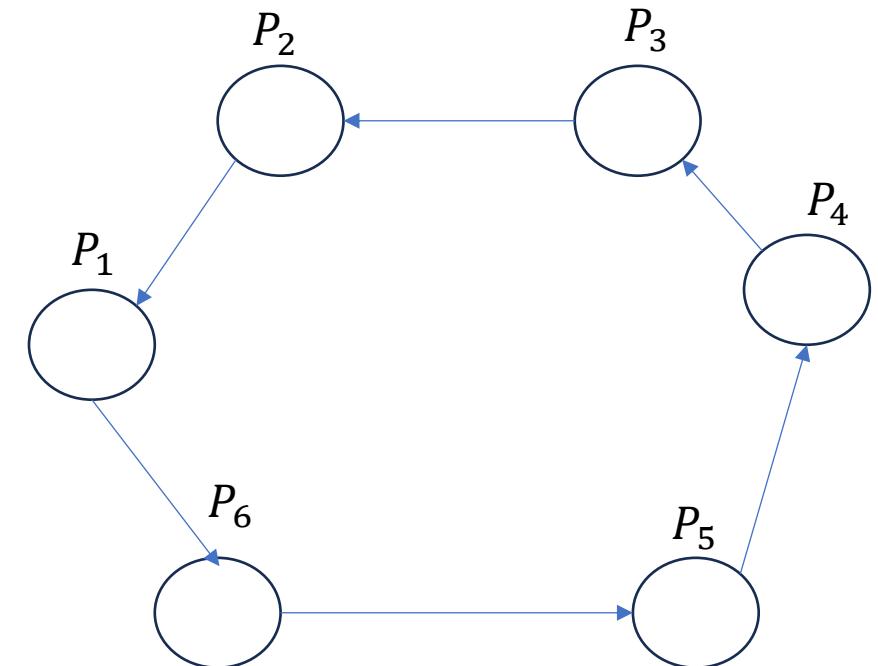
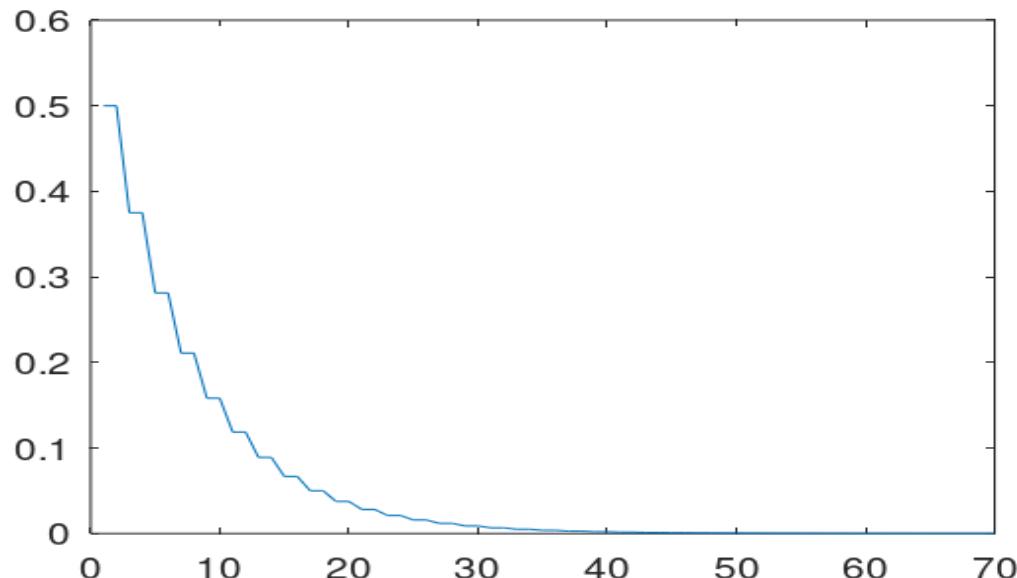
$$\begin{array}{cccccc} 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \end{array}$$



Exemplu

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$



$$V(t) = \max(x(t)) - \min(x(t))$$

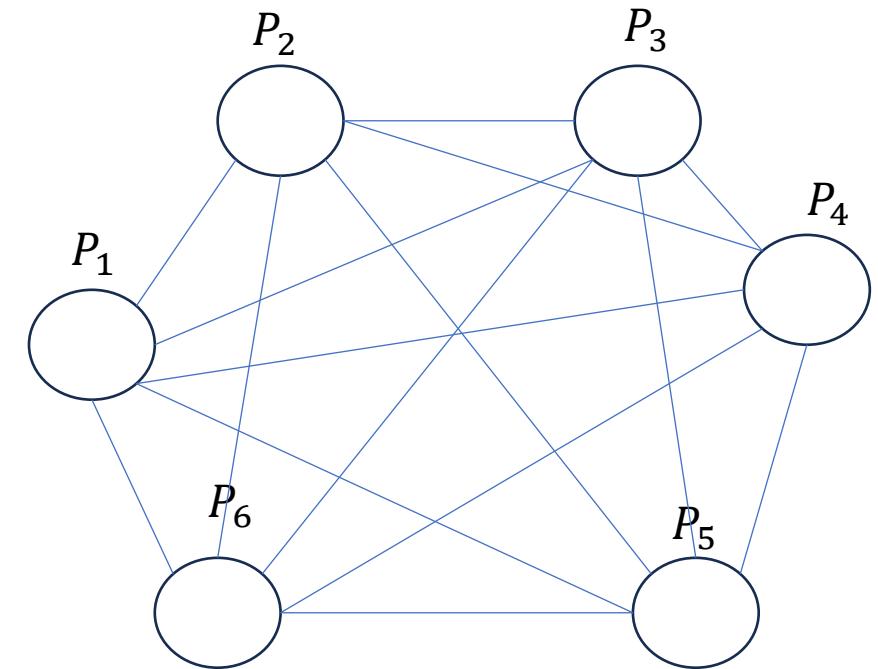
Plot: $V(t)$ vs. t

Exemple

Pentru un graf de tip clică matricea asociată este:

$$A = \begin{bmatrix} 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \end{bmatrix}$$

$$x(1) = x^*$$

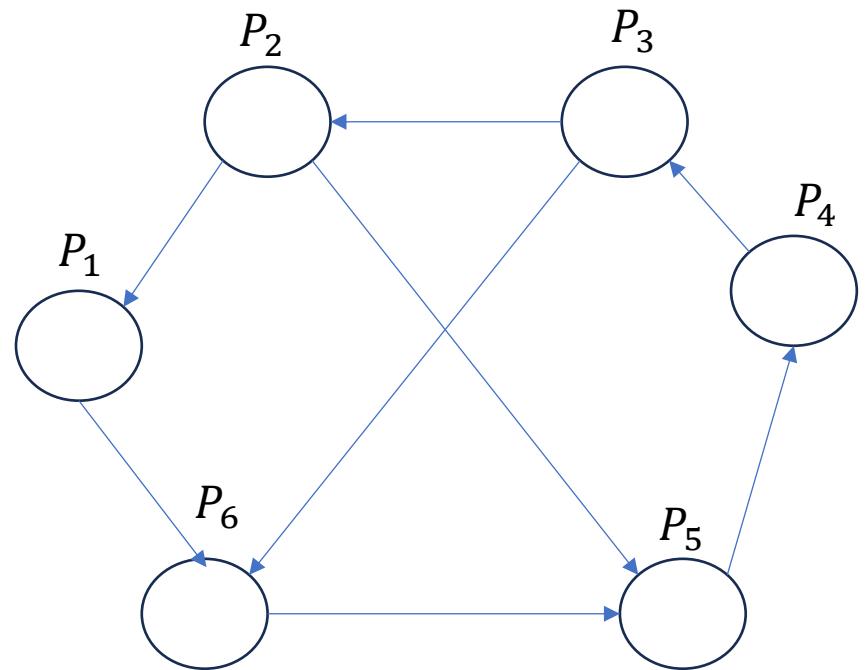


- Convergență într-un singur pas

Exemplu

Pentru graful alăturat matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 1/3 & 0 & 0 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 0 & 0 & 1/3 \end{bmatrix}$$



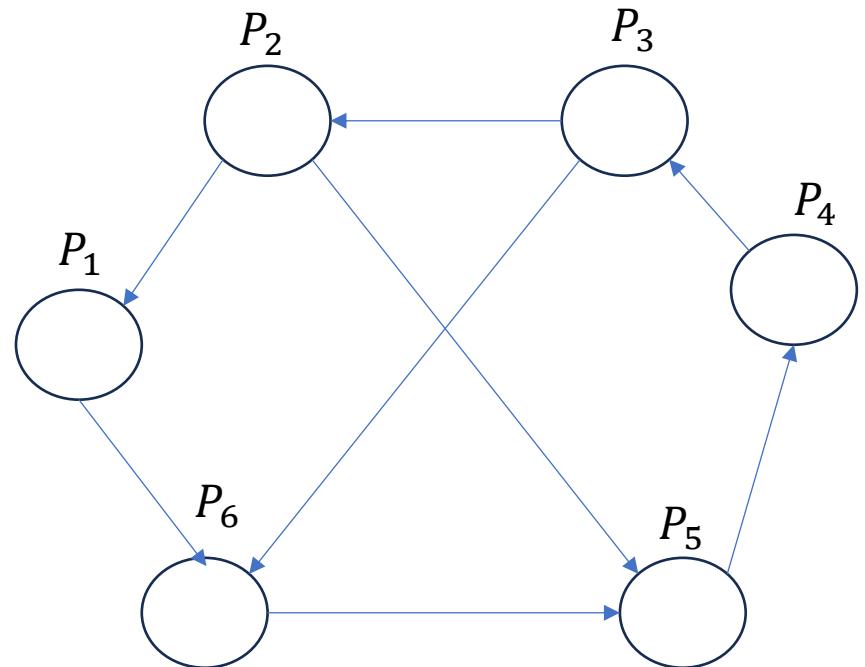
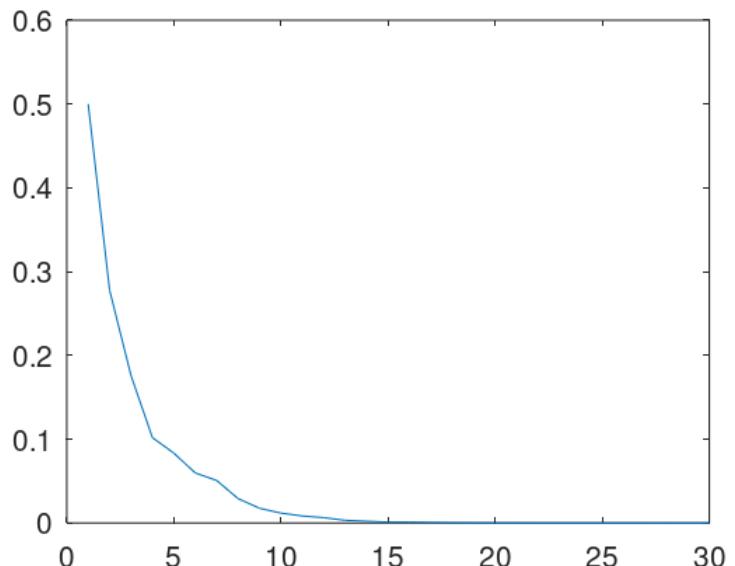
A^{30} :

0.060607	0.181818	0.242425	0.242424	0.181817	0.090909
0.060606	0.181818	0.242424	0.242425	0.181818	0.090909
0.060606	0.181818	0.242424	0.242424	0.181819	0.090909
0.060606	0.181818	0.242424	0.242424	0.181818	0.090909
0.060606	0.181818	0.242424	0.242424	0.181818	0.090909
0.060606	0.181818	0.242424	0.242424	0.181818	0.090909

Exemplu

Pentru graful alăturat matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 1/3 & 0 & 0 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 0 & 0 & 1/3 \end{bmatrix}$$



$$V(t) = \max(x(t)) - \min(x(t))$$

Plot: $V(t)$ vs. t

Defekte

Defecte

- Un număr sporit de noduri implică o probabilitate de defecte în creștere.
- Gravitatea defectului depinde de aplicație: sistem de control al traficului aerian vs sistem de gaming.
- Sursele defectelor la nivel de nod:
 - Erori: design, fabricație, programare
 - Accidente fizice
 - Condiții de mediu dure
 - Date de intrare neașteptate
 - Etc.

Modelarea defectelor

Pentru a evita restartarea algoritmilor de fiecare dată când apare un defect (sau adăugarea permanentă de resurse), sunt necesare *schemele (algoritmii) tolerante la defecte*.

Principalele modele:

- Defect *Crash*
- Defect *Bizantin*

Nodurile care nu suferă defect se numesc noduri **corecte**.

Redundanță

- Redundanță informației:
 - Adaugă extra biți pentru recuperarea datelor; tehnici ECC, coduri Hamming; checksum / CRC – pentru detectia alterării mesajelor.
- Redundanță de timp:
 - O acțiune executată va fi repetată dacă este necesar (E.g. Re-transmiterea pachetelor, tranzacții atomice, etc)
- Redundanță fizică:
 - Echipament extra adăugat în sistem pentru a tolera eventualele defecte.
 - Două moduri de organizare:
 - a) Active replication
 - b) Primary backup

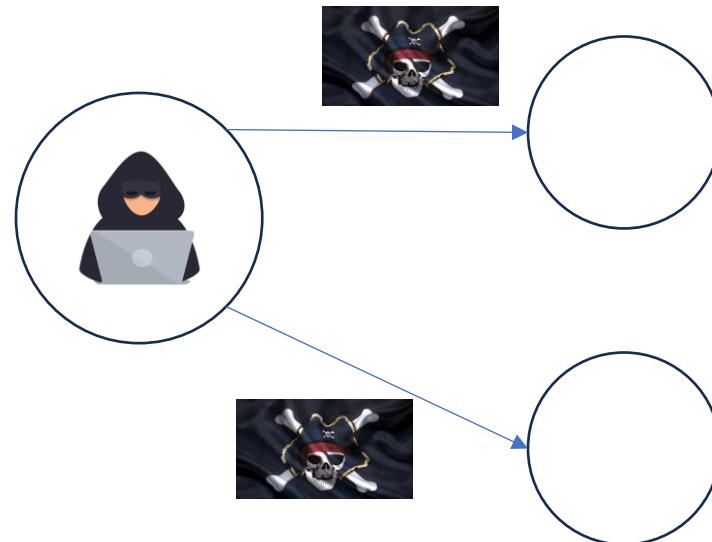
Defectul Crash

- La momentul defectului, nodul:
 - Oprescă execuția locală a iterațiilor
 - Nu primește mesaje
 - Nu trimit mesaje
- În perspectiva cea mai simplistă: nodul nu reia activitatea niciodată.
- Sub-clase de Crash:
 - Crash-stop
 - Omisiune de mesaje
 - Crash cu revenire

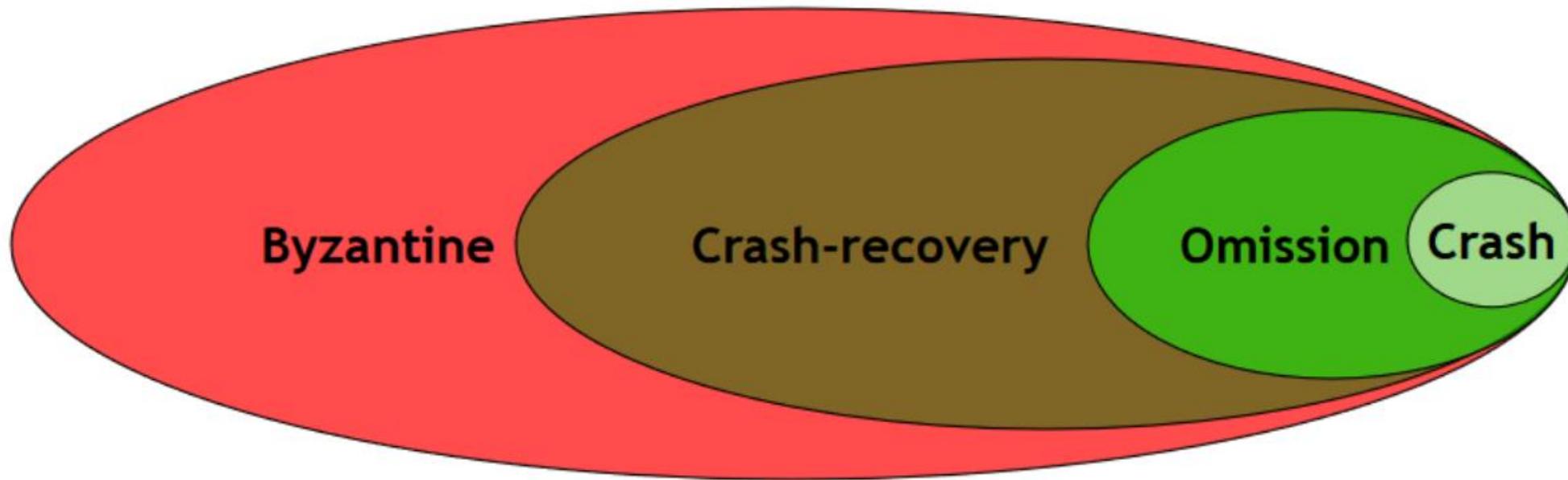
Defect Bizantin

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

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



Ierarchie



Consens distribuit (cu procese defecte)

Starea (valoarea) uniformă a nodurilor unui sistem distribuit.

Condiția de acord: Nu există două procese 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 *correct* din sistem va decide eventual la un moment de timp.

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

Consens distribuit (cu procese defecte)

Ipoteze:

- Graf complet (nedirectat)
- Un nod poate fi corect sau defect *Crash* (încetează funcționarea)
- Dacă un nod intră în starea de defect, rămâne în ea până la finalul algoritmului

Urmărim atingerea consensului distribuit sub maxim $s \geq 0$ defecte *Crash*.

Algoritm **FloodSet($f()$)**:

M_i : - int id (id propriu)
- int v (token, inițial egal cu x_i)
- funcție obiectiv $f()$
- int t , integer, inițial 0

Funcție transformare nod i ():

1. **Bcast**($< x_i(0), id_i >$)
2. Fie U mulțimea mesajelor $< v_j, id_j >$ primite restul nodurilor
3. $M_i(t+1) = M_i(t) \cup U$
4. Fie $V_i(t+1)$ mulțimea valorilor v_j din $M_i(t+1)$
5. Fie $I_i(t+1)$ mulțimea id-urilor id_j din $M_i(t+1)$
6. **Return** $f(V_i(t))$

Algoritm FloodSet pentru clică

Algoritm **FloodSet**($f()$):

- M_i : - int id (id propriu)
- int v (token, inițial egal cu x_i)
- funcție obiectiv $f()$
- int t , integer, inițial 0

Funcție transformare nod i ():

1. **Bcast**($< x_i(0), id_i >$)
2. Fie U mulțimea mesajelor $< v_j, id_j >$ primite restul nodurilor
3. $M_i(t+1) = M_i(t) \cup U$
4. Fie $V_i(t+1)$ mulțimea valorilor v_j din $M_i(t+1)$
5. Fie $I_i(t+1)$ mulțimea id-urilor id_j din $M_i(t+1)$
6. **Return** $f(V_i(t))$

$j \in \mathcal{N}_i^-$ cade la momentul t , atunci $M_j(t)$ nu va ajunge la P_i la momentul $t + 1$

$V_i \neq V_k$

Algoritm FloodSet s –robust (clică, defect Crash)

Algoritm **FloodSet**($f, x(0), s$):

- M_i : - int id (id propriu)
- int v (token, inițial egal cu x_i)
- funcție obiectiv $f()$
- int t , integer, inițial 0

Funcție transformare nod i ():

1. **Bcast**($M_i(t)$)
2. Fie U mulțimea mesajelor $\langle v_j, id_j \rangle$ primite restul nodurilor
3. $M_i(t+1) = M_i(t) \cup U$
4. Fie $V_i(t+1)$ mulțimea valorilor v_j din $M_i(t+1)$
5. Fie $I_i(t+1)$ mulțimea id-urilor id_j din $M_i(t+1)$
6. **If** ($t > s + 1$):
 1. **Return** $f(M_i(t))$
7. $t := t + 1$

- Paradigma de "robustificare" a algoritmilor distribuiți
- Se realizeaza $s + 1$ iteratii (s explicit)
- **Lemma 1.** Dacă există o iterație t în care nu există defect, atunci $M_i(t) = M_j(t)$ pentru orice noduri i și j corecte la momentul t .
- **Lemma 2.** Dacă $M_i(t) = M_j(t)$ pentru orice noduri i și j corecte. Atunci pentru orice $t \leq t' \leq f + 1$ avem $M_i(t') = M_j(t')$ pentru orice noduri i și j corecte la momentul t' .
- **Teorema.** FloodSet s –robust rezolvă problema de consens pentru defecte de tip *Crash*.
- Principalul argument: avem s defecte, de aceea după $s + 1$ iterații va exista cel puțin o iterație t în care nu există defect. Lemma 1 implică $M_i(t) = M_j(t)$ pentru orice noduri i și j corecte la momentul t . Lemma 2 implică $M_i(s + 1) = M_j(s + 1)$ pentru orice noduri i și j corecte la momentul $s + 1$.

Algoritm FloodSet s –robust (clică, defect Crash)

Algoritm **FloodSet**($f, x(0), s$):

- M_i : - int id (id propriu)
- int v (token, inițial egal cu x_i)
- funcție obiectiv $f()$
- int t , integer, inițial 0

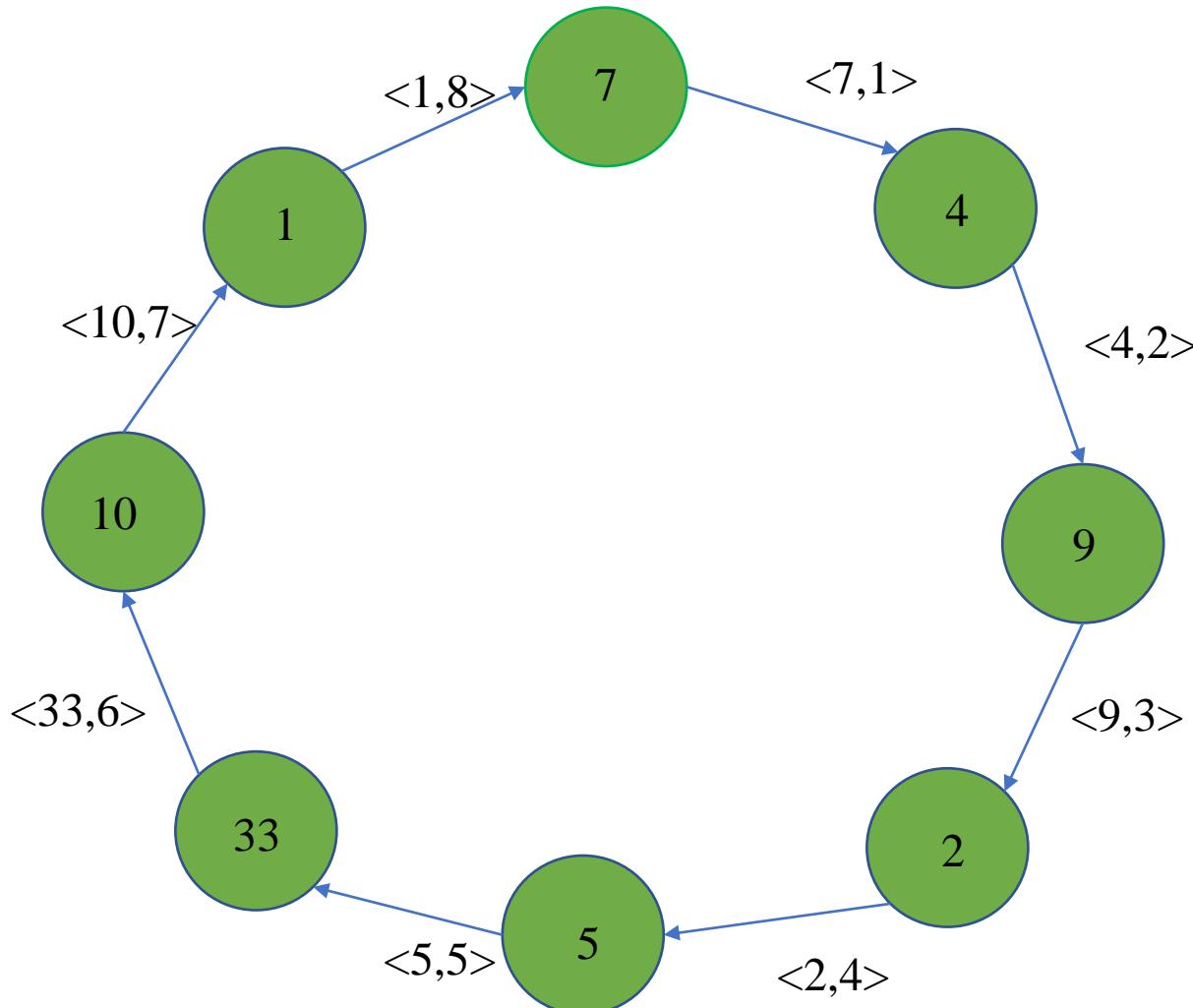
Funcție transformare nod i ():

- 1. Bcast**($M_i(t)$)
2. Fie U mulțimea mesajelor $\langle v_j, id_j \rangle$ primite restul nodurilor
3. $M_i(t+1) = M_i(t) \cup U$
4. Fie $V_i(t+1)$ mulțimea valorilor v_j din $M_i(t+1)$
5. Fie $I_i(t+1)$ mulțimea id-urilor id_j din $M_i(t+1)$
- 6. If** ($t > s + 1$):
 - 1. Return** $f(M_i(t))$
7. $t := t + 1$

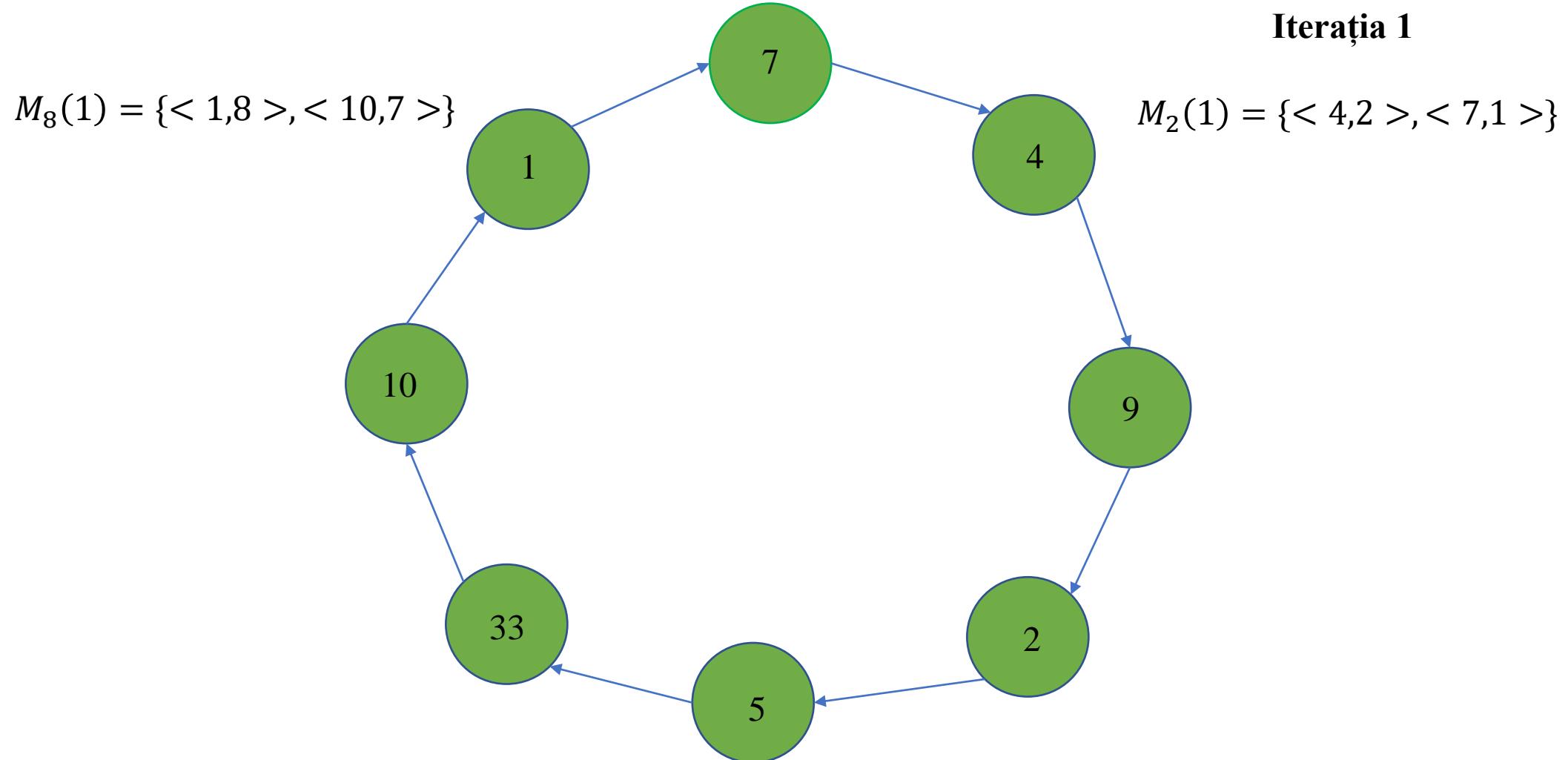
- Operatia de Broadcast costa n mesaje
- Complexitate timp: $s + 1$
- Complexitate mesaj: $(s + 1)n$
- Rezultat marginie inferioara (informal)[Lynch]: orice algoritm s-robust la defecte Crash executa minim $s+1$ iteratii
- Cum generalizăm pentru grafuri conexe generale (nu neapărat complete)?

Algoritm FloodSet s – robust

Iterația 1

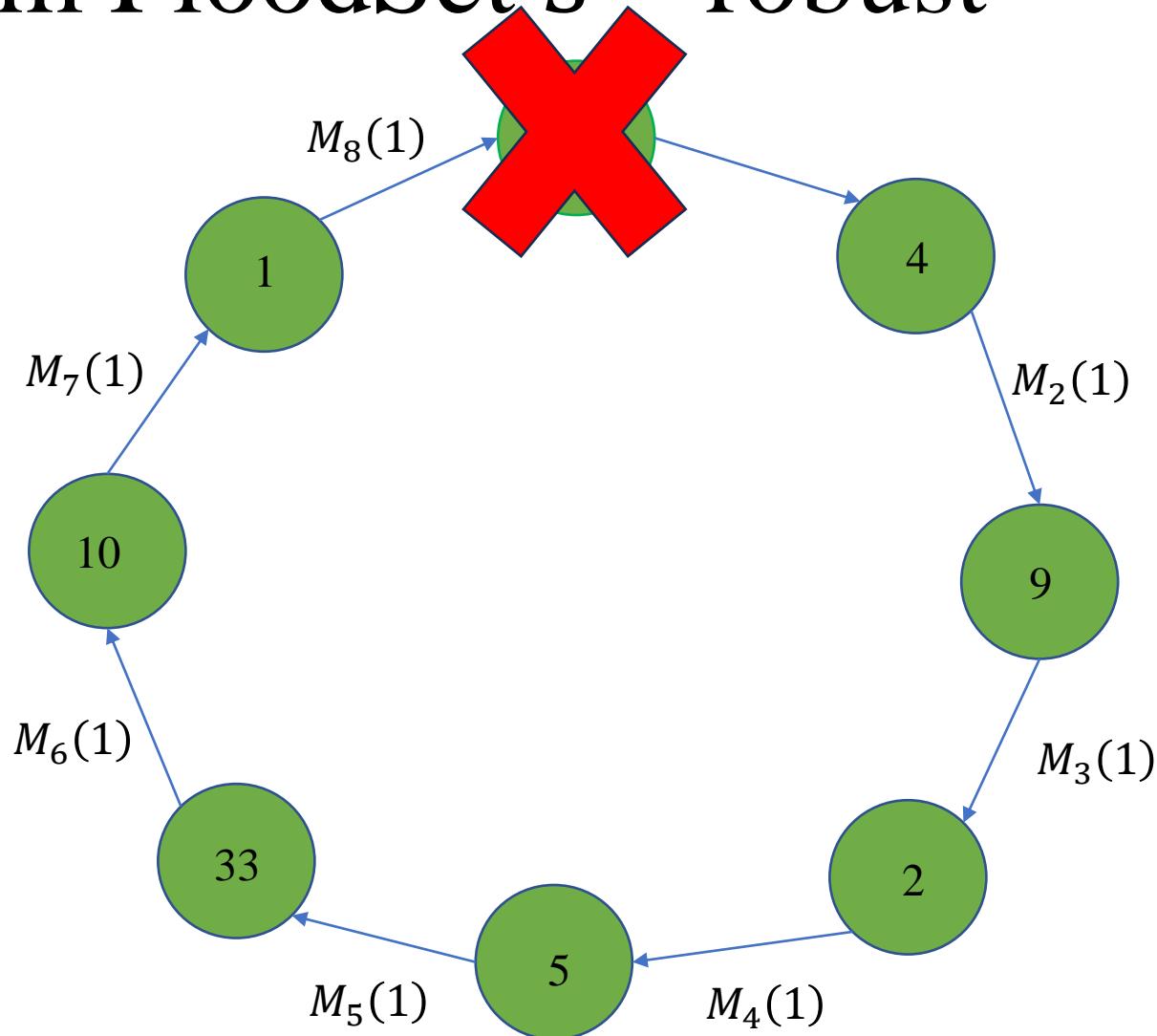


Algoritm FloodSet s –robust



Algoritm FloodSet s – robust

Iterația 2

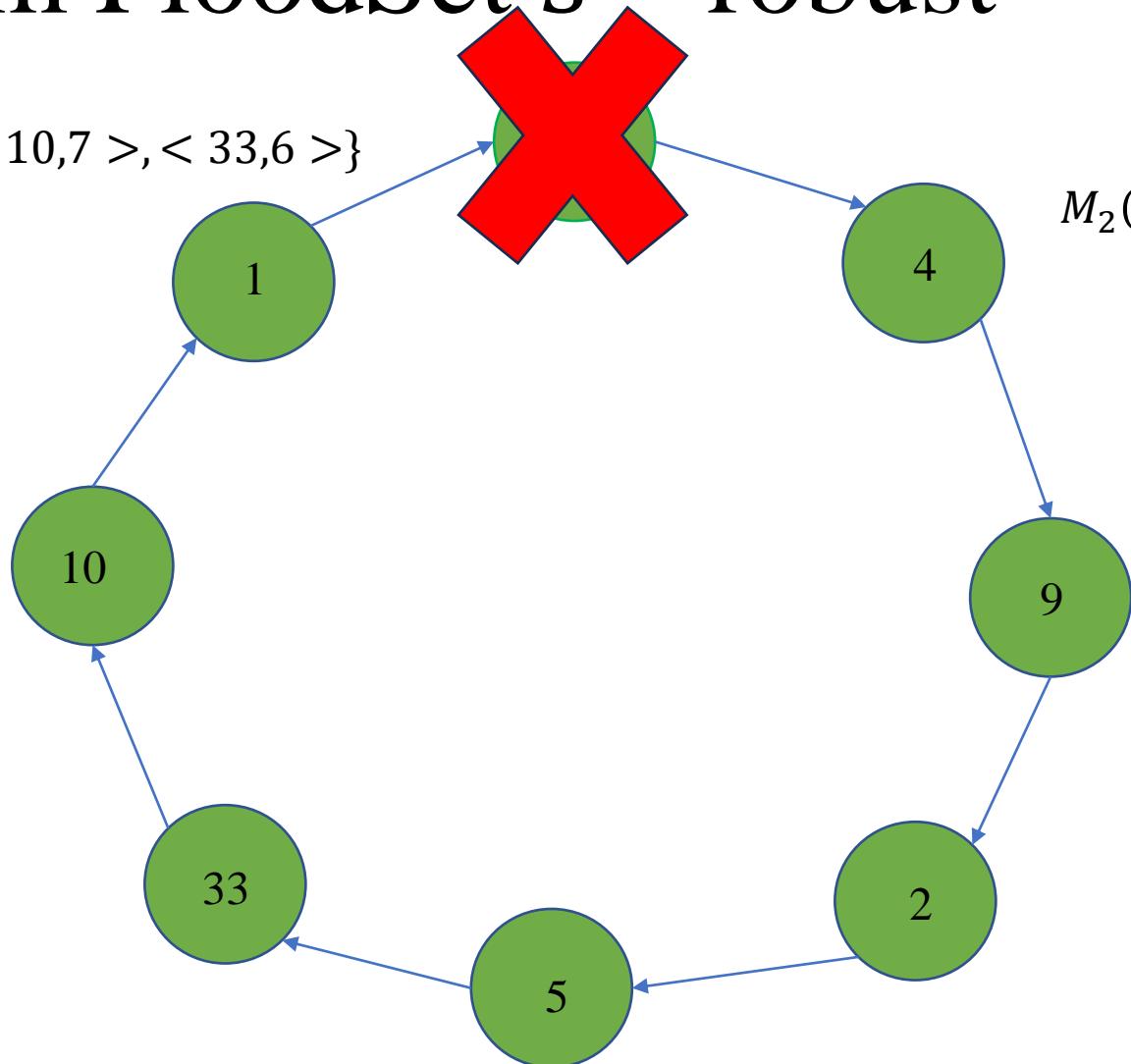


Algoritm FloodSet s – robust

$$M_8(1) = \{<1,8>, <10,7>, <33,6>\}$$

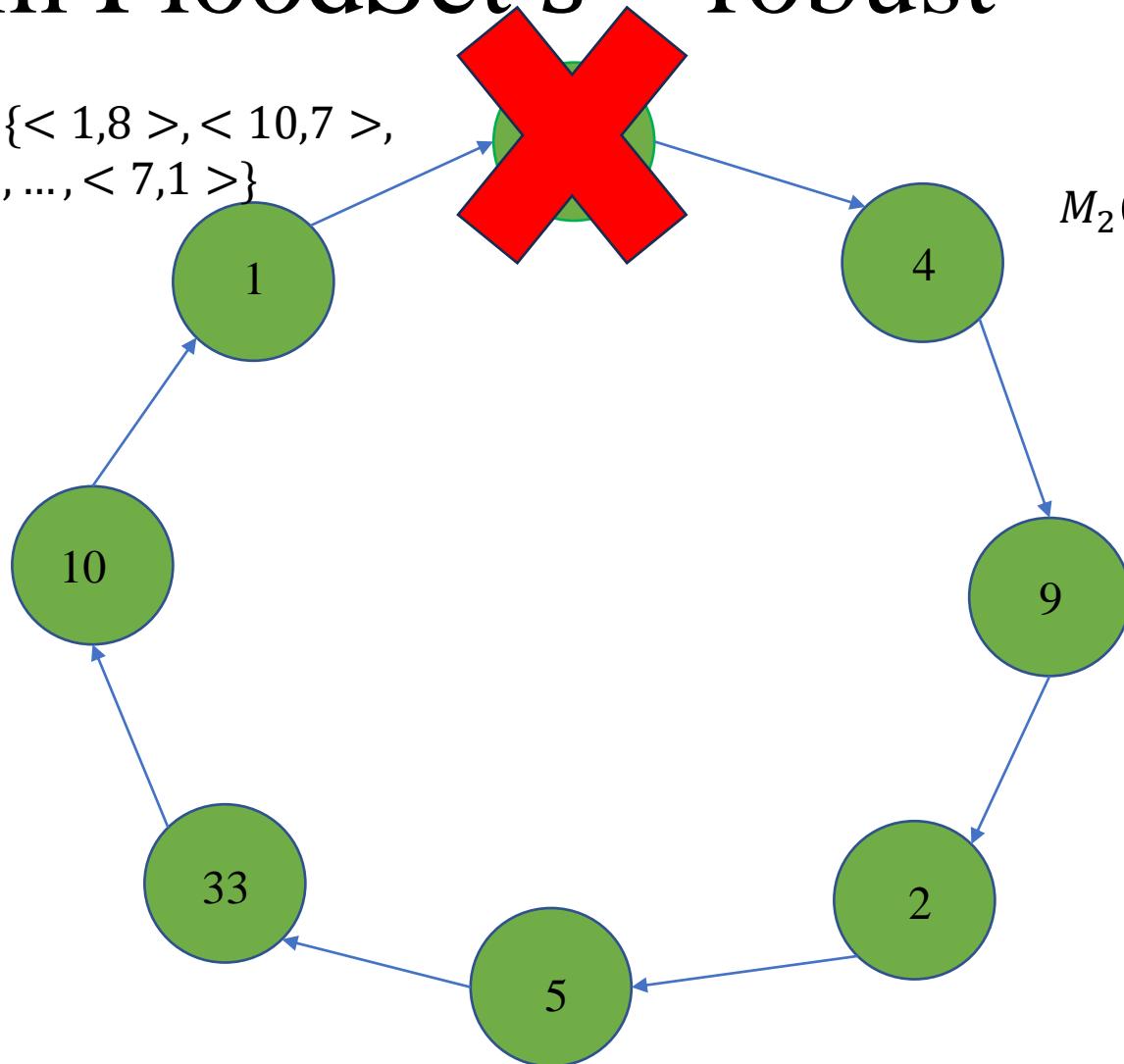
Iterația 2

$$M_2(1) = \{<4,2>, <7,1>\}$$



Algoritm FloodSet s – robust

$$M_8(7) = \{<1,8>, <10,7>, \\ <33,6>, \dots, <7,1>\}$$



Iterația 7

$$M_2(7) = \{<4,2>, <7,1>\}$$

Consens distribuit (cu procese defecte)

Ipoteze:

- Graf ~~complet~~ conex (nedirectat). Conectivitatea grafului G , $\text{conn}(G)$ = numărul minim de noduri care, o dată eliminate din graf, va rezulta un graf neconectat.
- $s < \text{conn}(G)$
- Un nod poate fi corect sau defect *Crash* (încetează funcționarea)
- Dacă un nod intră în starea de defect, rămâne în ea până la finalul algoritmului

Urmărim atingerea consensului distribuit sub maxim $s \geq 0$ defecte *Crash*.

Algoritm FloodSet s –robust (conex, defect Crash)

Algoritm **FloodSet**($f()$):

- M_i : - int id (id propriu)
- int v (token, inițial egal cu x_i)
- funcție obiectiv $f()$
- int t , integer, inițial 0

Funcție transformare nod i ():

1. Fie U mulțimea mesajelor $\langle v_j, id_j \rangle$ primite de la \mathcal{N}_i^-
2. $M_i(t+1) = M_i(t) \cup U$
3. Fie $V_i(t+1)$ mulțimea valorilor v_j din $M_i(t+1)$
4. Fie $I_i(t+1)$ mulțimea id-urilor id_j din $M_i(t+1)$
5. **If** ($t > (s + 1)diam$):
 1. **Return** $f(M_i(t))$
6. **Else**: $\text{send}(M_i(t+1), \mathcal{N}_i^+)$
7. $t := t + 1$

- Ipoteza $s < conn(G)$ garantează că graful rezultat în urma defectelor rămâne conex.
- Se realizează $s + 1$ seturi de $diam(G)$ iterații.
- Convergența folosește aceleași argumente ca în cazul clicii; în principal, după $s + 1$ seturi de $diam(G)$ iterații, există cel puțin unul în care niciun nod nu are defect. Însă $diam(G)$ sunt suficiente pentru a atinge consensul între nodurile corecte.

Sisteme și algoritmi distribuiți

Curs 6

Valori și vectori proprii

Definiție. Valorile proprii (*eigenvalues*) ale matricii $A \in R^{n \times n}$ sunt date de n rădăcini ale polinomului caracteristic $p(z) = \det(zI_n - A)$. Multimea acestor valori se numește spectrul matricii A și este notat cu:

$$\lambda(A) = \{z : \det(zI_n - A) = 0\}.$$

Definiție. Pentru $\lambda \in \lambda(A)$ numim vectorii nenuli $x \in C^n$ care satisfac $Ax = \lambda x$ vectori proprii (*eigenvectors*). Mai exact x este vector propriu *la dreapta* dacă satisfacă:

$$Ax = \lambda x$$

și vector propriu *la stânga* dacă satisfacă:

$$x^H A = x^H \lambda.$$

Un vector propriu definește un subspațiu 1-dimensional care este invariant la premultiplicarea cu A .

Reamintim: pentru $x \in C^n$, x^H reprezintă vectorul x transpus și conjugat.

Valori și vectori proprii - exemple

În cazul matricilor diagonale (triunghiulare), valorile proprii sunt elementele diagonalei, e.g.

$$\lambda(I_n) = \{1, \dots, 1\}.$$

Dacă $A = A^T$, atunci subspațiul v. p. la dreapta = subspațiul v. p. la stânga.

Când calculăm un vectorul propriu asociat unei valori proprii fixate, în general ne interesează doar vectorul de pe sfera unitate.

Exemplu: Calculați vectorul propriu (la stânga și dreapta) asociat valorii proprii 1

ai matricii $A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$.

Valori și vectori proprii (matrici stohastice)

Teorema Perron-Frobenius. Dacă matricea $A \in R^{n \times n}$ este stochastică (pe linii) și ireductibilă atunci: vectorul propriu la stânga $w \in R^n$ satisfacă $w \geq 0$ și

- 1) $\rho(A) = 1$ este simplă. (Valoarea Perron-Frobenius)
- 2) Vectorii proprii asociati lui $\rho(A)$ au componente pozitive.
- 3) Fie w v. p. la stânga asociat lui $\rho(A)$ atunci $\lim_{t \rightarrow \infty} A^t = 1w^T$. (Proiecția Perron)

Matrice ireductibilă. Matricea A este *ireductibilă* dacă nu este similară via permutări cu o matrice bloc superior triunghiulară.

Dacă matricea A este matricea de adiacență asociată unui *graf (tare) conex*, atunci A este ireductibilă.

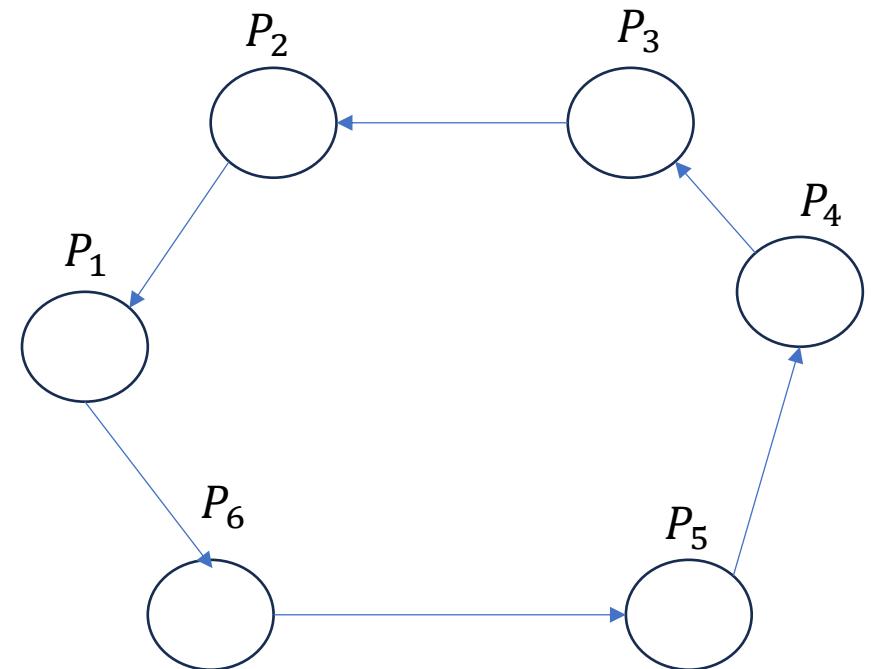
Din cursul trecut

Pentru un inel de dimensiune $n = 6$, matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 1/2 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}$$

A^70 :

$$\begin{array}{cccccc} 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \end{array}$$

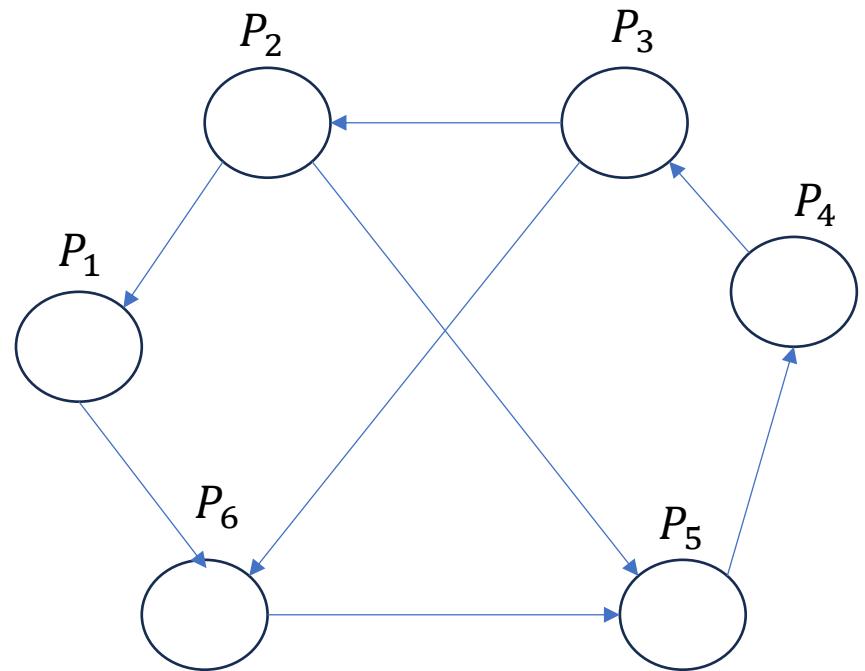


$$= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \end{bmatrix}$$

Exemplu

Pentru graful alăturat matricea asociată este:

$$A = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 1/3 & 0 & 0 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 0 & 0 & 1/3 \end{bmatrix}$$



A^{30} :

0.060607	0.181818	0.242425	0.242424	0.181817	0.090909
0.060606	0.181818	0.242424	0.242425	0.181818	0.090909
0.060606	0.181818	0.242424	0.242424	0.181819	0.090909
0.060606	0.181818	0.242424	0.242424	0.181818	0.090909
0.060606	0.181818	0.242424	0.242424	0.181818	0.090909
0.060606	0.181818	0.242424	0.242424	0.181818	0.090909

$$= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} [0.061 \quad 0.182 \quad 0.243 \quad 0.243 \quad 0.182 \quad 0.091]$$

Algoritm Flooding pentru consens

Algoritm Flooding de medie prezentat anterior se exprimă recurrent prin actualizarea liniară:

$$x_i(t+1) = a_{ii}x_i(t) + \sum_{j \in \mathcal{N}_i^-} a_{ij}x_j(t) \quad \forall i$$

Deci $x_i(t+1) = a_i^T x(t)$, iar dinamica stărilor sistemului este:

$$x(t+1) = Ax(t),$$

unde

$$A = \begin{bmatrix} a_1^T \\ \dots \\ a_n^T \end{bmatrix} \in R^{n \times n}, x(t) = \begin{bmatrix} x_1(t) \\ \dots \\ x_n(t) \end{bmatrix} \in R^n.$$

Algoritm Flooding pentru consens

Din dinamică stărilor

$$x(t+1) = Ax(t),$$

se observă ușor:

$$x(t) = A^t x(0),$$

de aceea convergența depinde total de comportamentul matricii A^t (implicit, doar de structura grafului).

Teorema. Dacă matricea A este *stochastică pe linii*, i.e. $a_{ii} + \sum_{j \in \mathcal{N}_i^-} a_{ij} = 1, a_{ij} \geq 0 \quad \forall j \in \mathcal{N}_i^- \cup \{i\}$, atunci se atinge consensul asimptotic:

$$x(t) \rightarrow c\mathbf{1} \text{ când } t \rightarrow \infty.$$

În plus, dacă matricea A este *stochastică pe coloane*, i.e. $\mathbf{1}^T A = A^T$, atunci valoarea de consens este

$$c = \frac{1}{n} \sum_{i=1}^n x_i(0).$$

Algoritm Flooding pentru consens

Fie v vectorul propriu la stânga al matricii A asociat valorii proprii 1, și iterația

$$x(t + 1) = Ax(t),$$

atunci

$$v^T x(t) = v^T A^t x(0) = v^T x(0).$$

Observație: Unghiul tuturor iterațiilor $x(t)$ față de v este constant pentru orice t .

De aceea, în cazul convergenței, la limită: $x(\infty) = \lim_{t \rightarrow \infty} A^t x(0) = c\mathbf{1}$ avem relația (din th. P-F)
 $v^T x(\infty) = c = v^T x(0),$

concluzionând că valoarea de consens este dată de $v^T x(0)$.

Pentru a răspunde la întrebarea:

Care este valoarea de consens a algoritmului de Flooding pe o topologie particulară?
este necesară calcularea vectorul propriu la stânga al matricii A asociat valorii proprii 1.

Consens distribuit (cu procese defecte)

Ipoteze:

- Graf ~~complet~~ conex (nedirectat). Conectivitatea grafului G , $conn(G)$ = numărul minim de noduri care, o dată eliminate din graf, va rezulta un graf neconectat.
- $s < conn(G)$
- Un nod poate fi corect sau defect *Crash* (încetează funcționarea)
- Dacă un nod intră în starea de defect, rămâne în ea până la finalul algoritmului

Urmărim atingerea consensului distribuit sub maxim $s \geq 0$ defecte *Crash*.

Algoritm Flooding pentru consens

Algoritm Flooding de medie prezentat anterior se exprimă recurrent prin actualizarea liniară:

$$x_i(t+1) = a_{ii}x_i(t) + \sum_{j \in \mathcal{N}_i^-} a_{ij}x_j(t) \quad \forall i$$

Deci $x_i(t+1) = a_i^T x(t)$, iar dinamica stărilor sistemului este:

$$x(t+1) = Ax(t),$$

unde

$$A = \begin{bmatrix} a_1^T \\ \dots \\ a_n^T \end{bmatrix} \in R^{n \times n}, x(t) = \begin{bmatrix} x_1(t) \\ \dots \\ x_n(t) \end{bmatrix} \in R^n.$$

Problemă

Fie graful $G = (V, E)$ cu matricea de adiacență ponderată $A \in R^{5 \times 5}$.

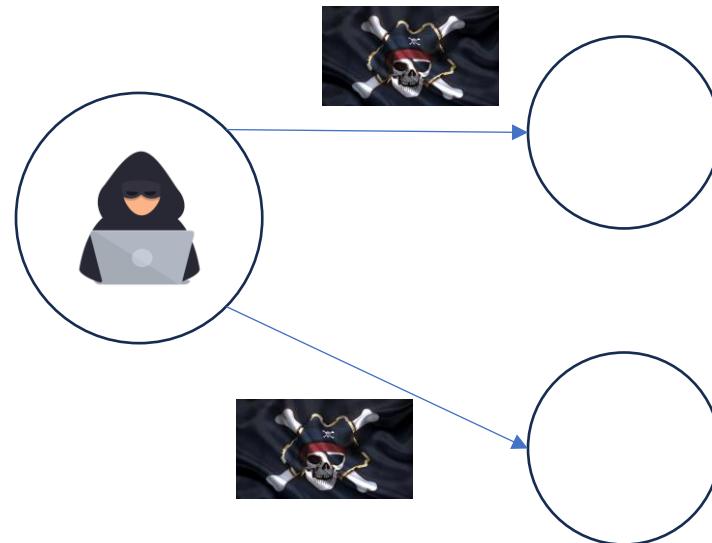
- a) Care este valoarea de consens asimptotic a algoritmului Flooding de medie?
- b) Presupunem că la momentul de timp $t = 3$, nodurile 2 și 3 suferă defect de tip *Crash*. Cum se schimbă valoarea de consens asimptotic a algoritmului Flooding de medie în acest caz?

Defecte bizantine

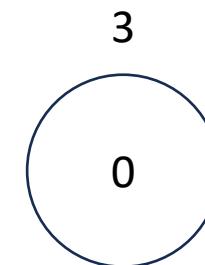
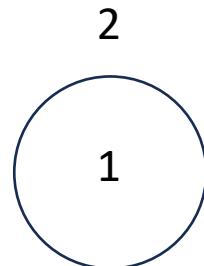
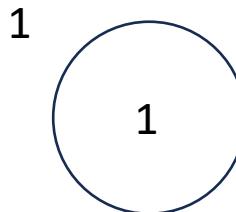
Defect Bizantin

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

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



Exemplu

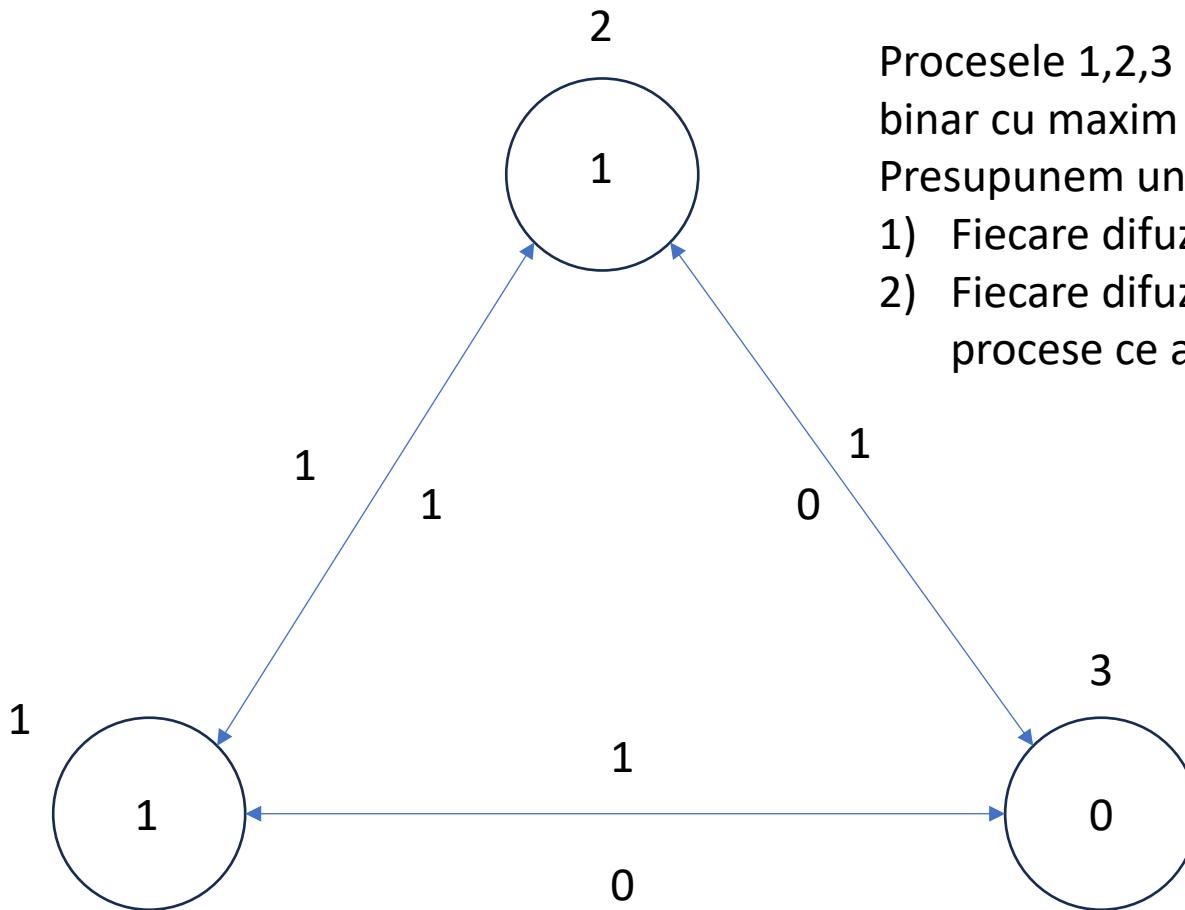


Procesele 1,2,3 rezolva problema de consens bizantin binar cu maxim un defect.

Presupunem un algoritm cu 2 runde:

- 1) Fiecare difuzeaza propria valoare initiala
- 2) Fiecare difuzeaza catre fiecare din celelalte procese ce a primit de la al treilea proces

Exemplu



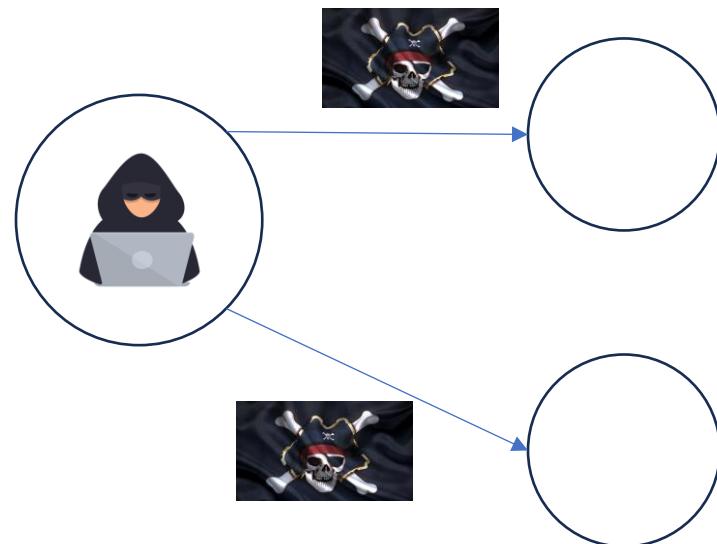
Procesele 1,2,3 rezolva problema de consens bizantin binar cu maxim un defect.

Presupunem un algoritm cu 2 runde:

- 1) Fiecare difuzeaza propria valoare initiala
- 2) Fiecare difuzeaza catre fiecare din celelalte procese ce a primit de la al treilea proces

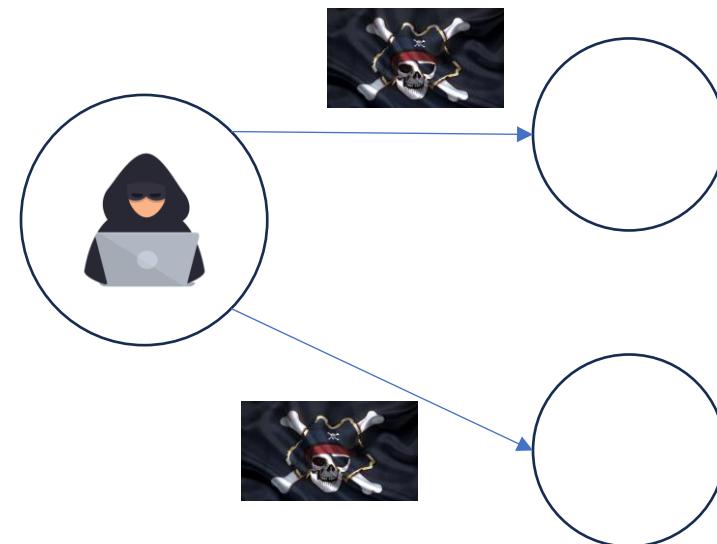
Defect Bizantin

Exemplul precedent stă la baza demonstrației faptul că $n > 3f$ procese sunt necesare pentru toleranța a f defecte bizantine.



Defect Bizantin

Algoritmii robusti la defecte bizantine se bazeaza pe operatii de broadcast speciale pentru a difuza informatica corecta in sistem.



Algoritmul ByzFlood

Sistem cu n procese și maxim s defecte (bizantine)

Urmărim consensul binar majoritar (distribuit): $x(0) \in \{0,1\}^n$

$$x_i^* = Maj(x(0)) = \begin{cases} 1, & \text{dacă } |\{i | x_i(0) = 1\}| \geq \frac{n}{2} + 1 \\ 0, & \text{dacă } |\{i | x_i(0) = 1\}| < \frac{n}{2} + 1 \end{cases}$$

Pentru început, considerăm graful sistemului *complet*.

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

Functie 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. $mult(t) = \text{numărul de apariții al } Majority(t)$

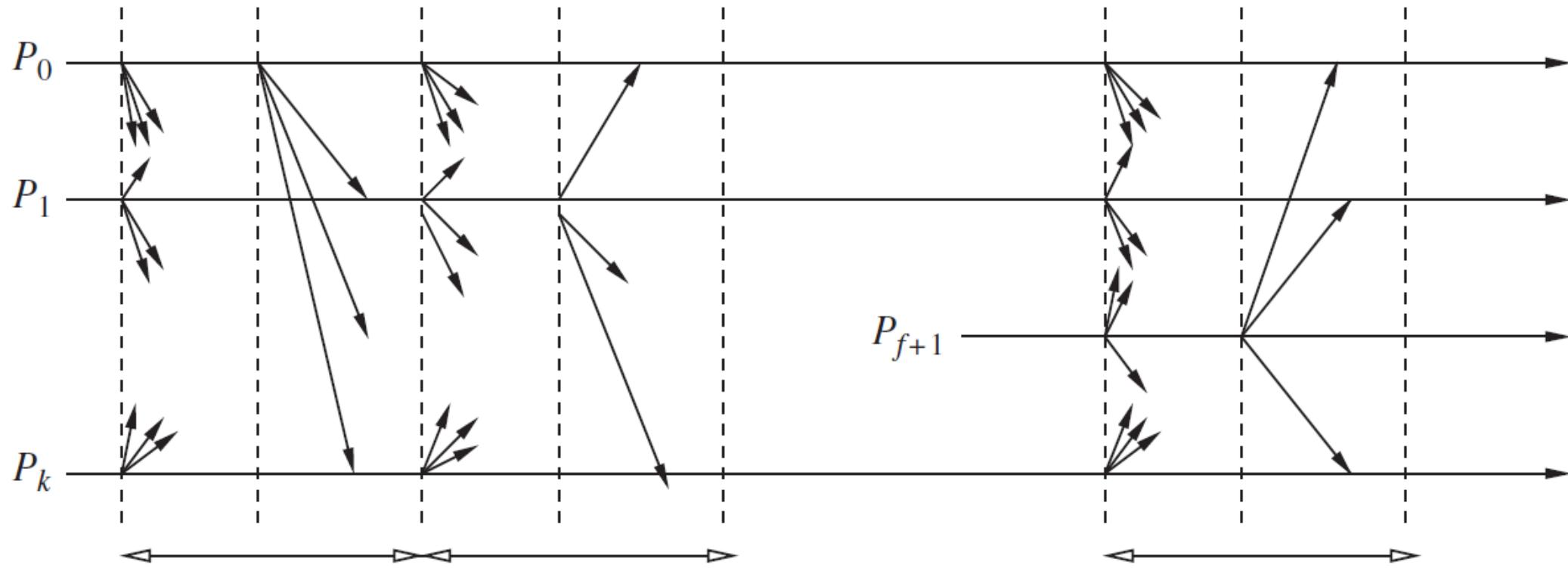
% Runda 2

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. $t := t + 1$

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_t sunt primit cel puțin $n/2$ voturi pentru aceeași valoare.

În cele 3 situații P_i și P_j ajung la consens.

Algoritmul ByzFlood



Sisteme și algoritmi distribuiți

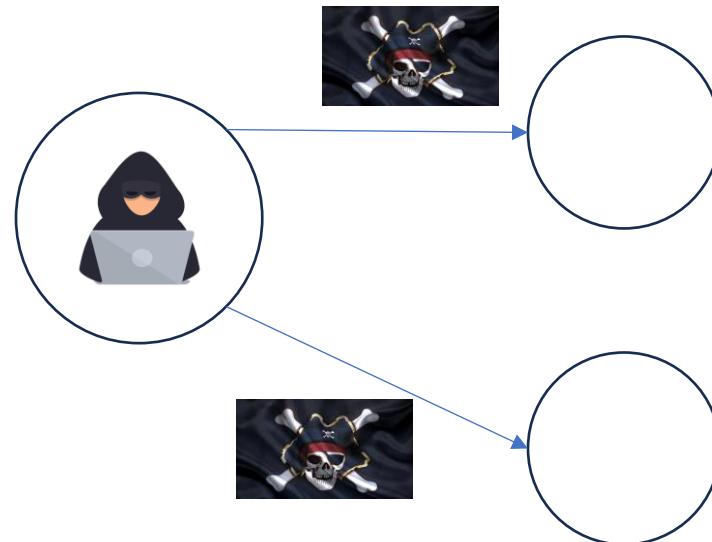
Curs 7

Defecțe bizantine (reluare)

Defect Bizantin

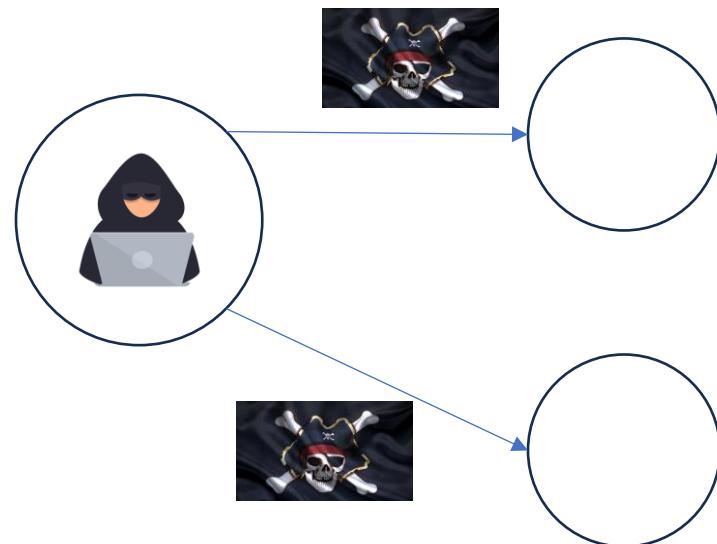
Sub defect bizantin nodurile se comportă malitios, perturbând activitatea întregului sistem (e.g. comportament arbitrar):

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



Defect Bizantin

Teorema [Lynch]. Într-un sistem distribuit sunt necesare $n > 3s$ procese pentru toleranță 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 *correct* din sistem va decide eventual la un moment de timp.

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

Consens distribuit (cu procese defecte)

Ipoteze:

- Sistem cu n procese și maxim s defecte (bizantine)
- Pentru început, considerăm graf *complet* al sistemului.
- Presupunem că numărul de noduri din sistem satisfacă: $n > 4s$.
- 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 ():

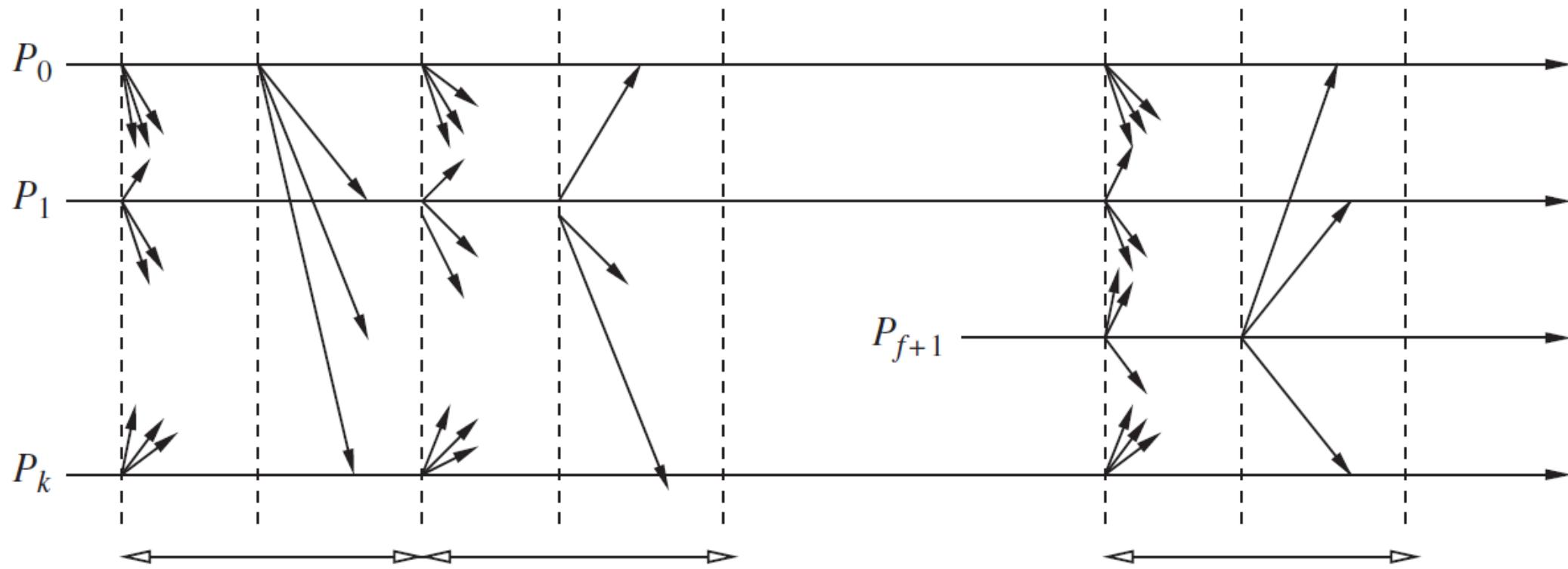
% Runda 1

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. $mult(t) = \text{numărul de apariții al valorii } Majority(t)$

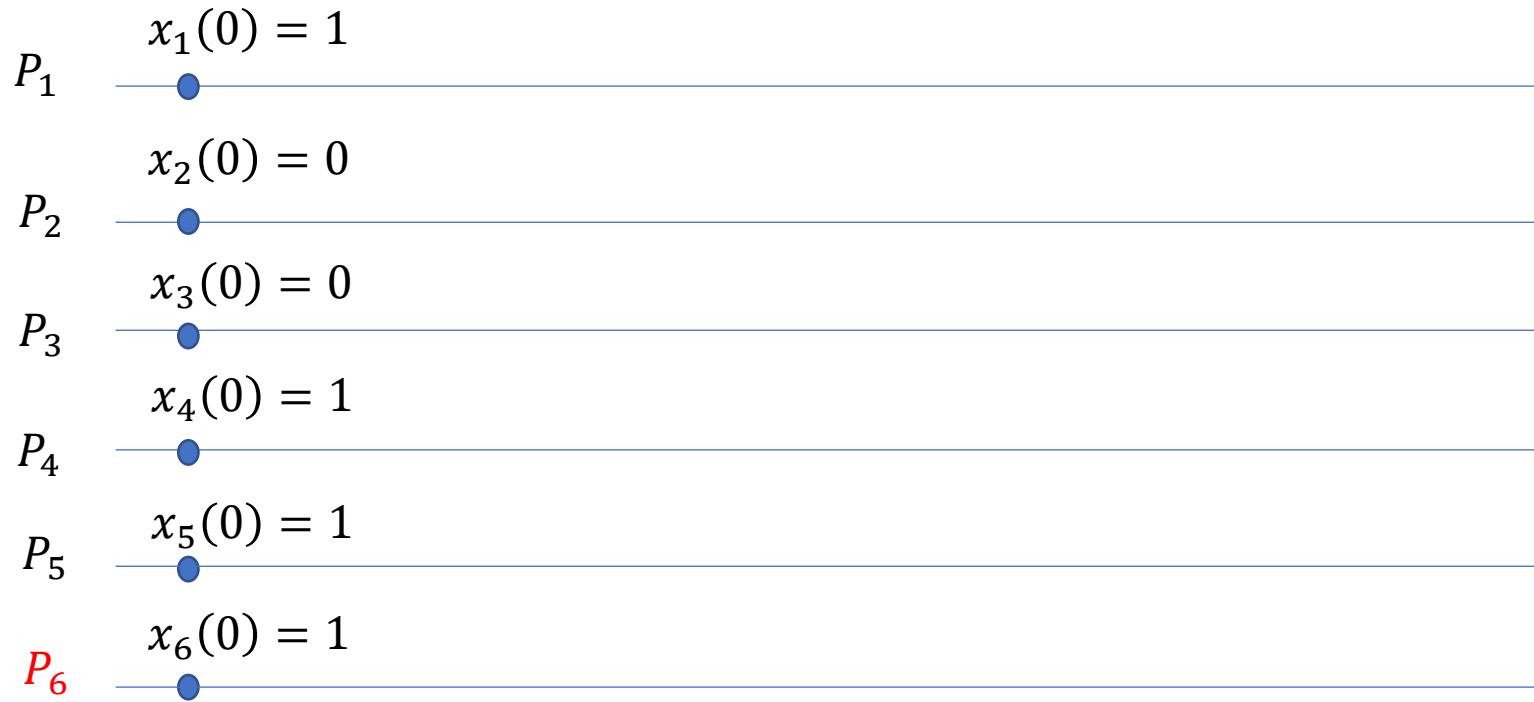
% Runda 2

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

Algoritmul ByzFlood



Exemplu ByzFlood($s = 1$)



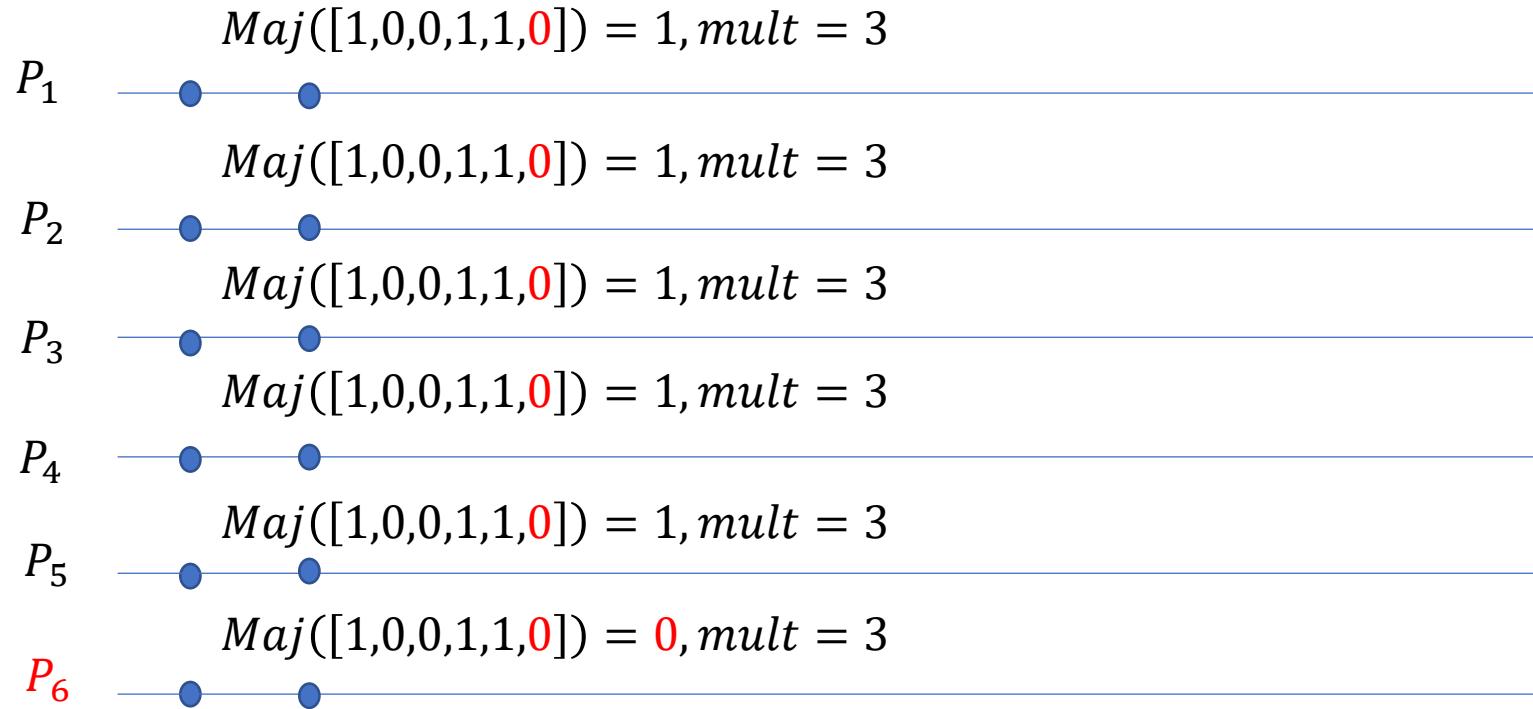
$t = 0 (n > 3s, P_6 \text{ defect arbitrar})$

Exemplu ByzFlood



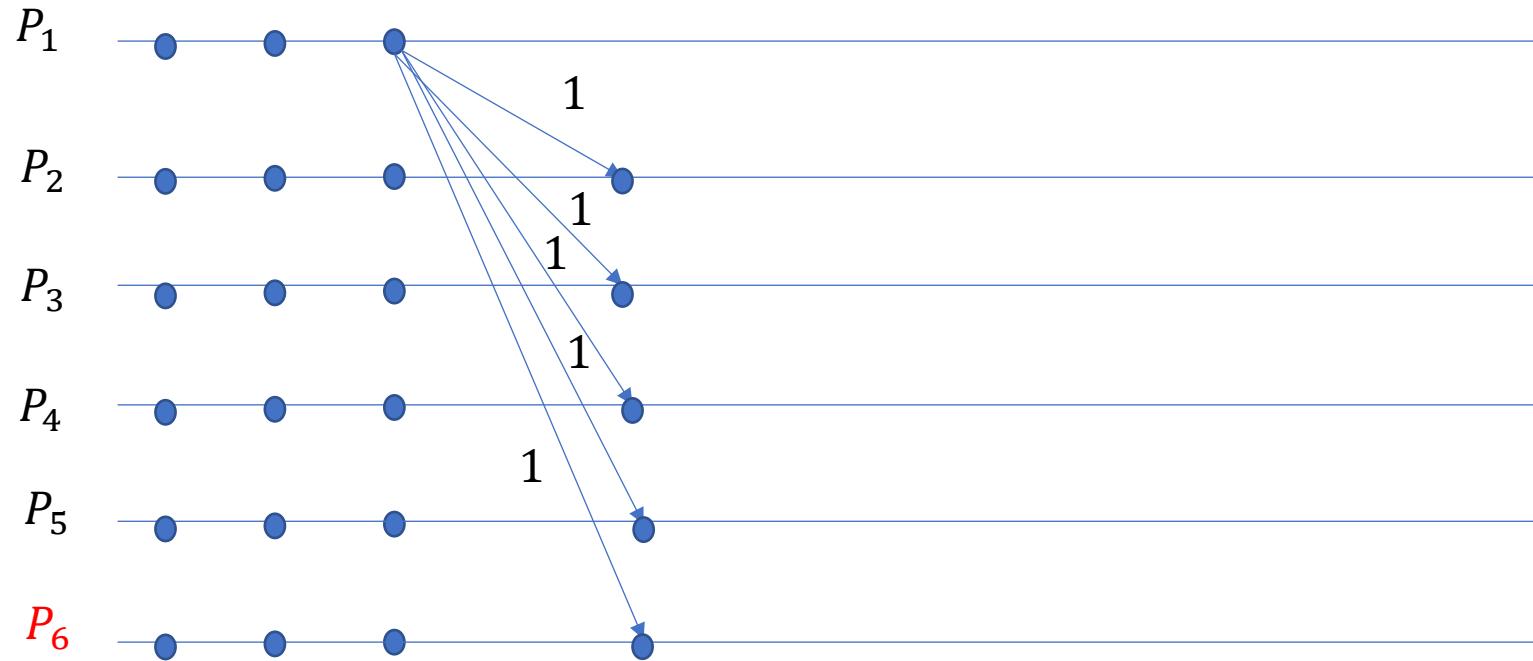
$t = 1: Bcast(x_i(0))$

Exemplu ByzFlood



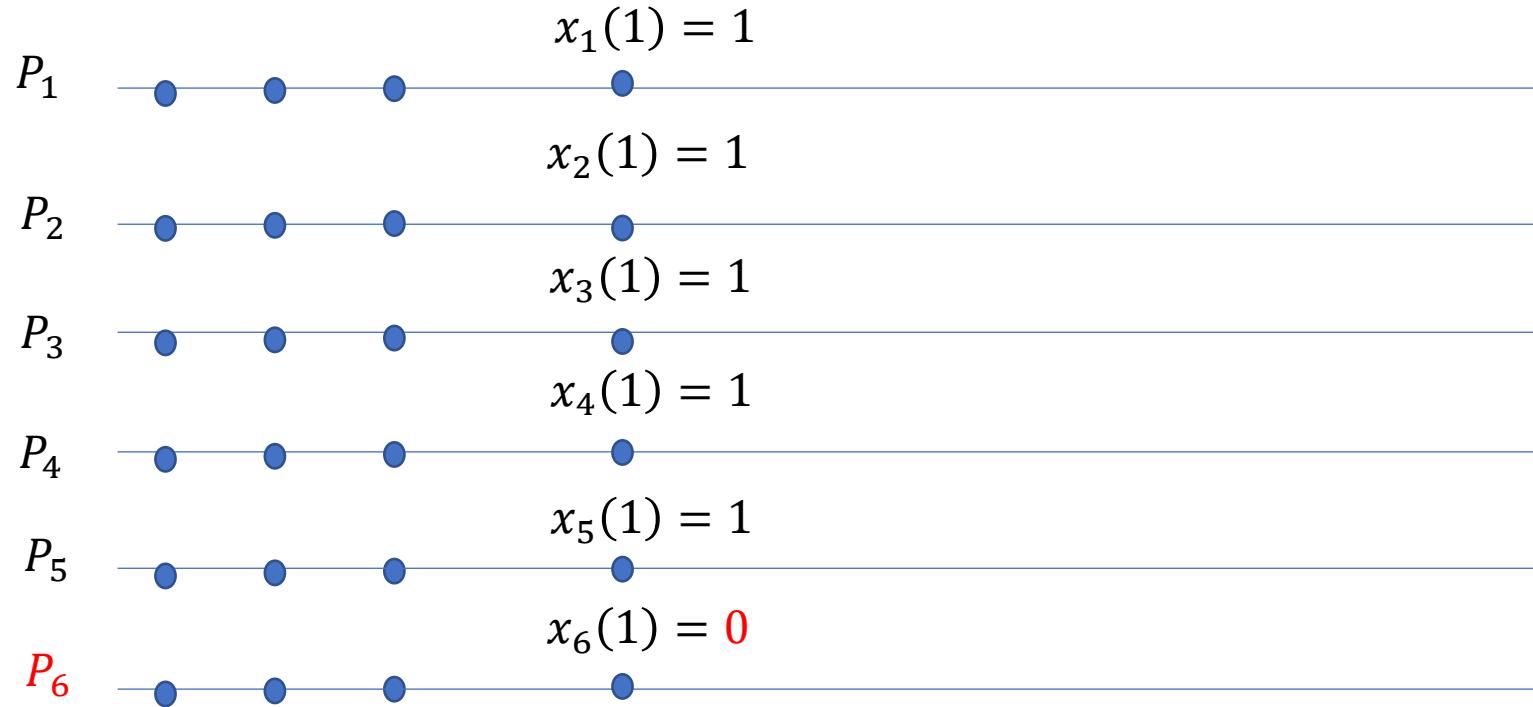
$t = 1 : \text{Calcul Maj}(U), mult$

Exemplu ByzFlood



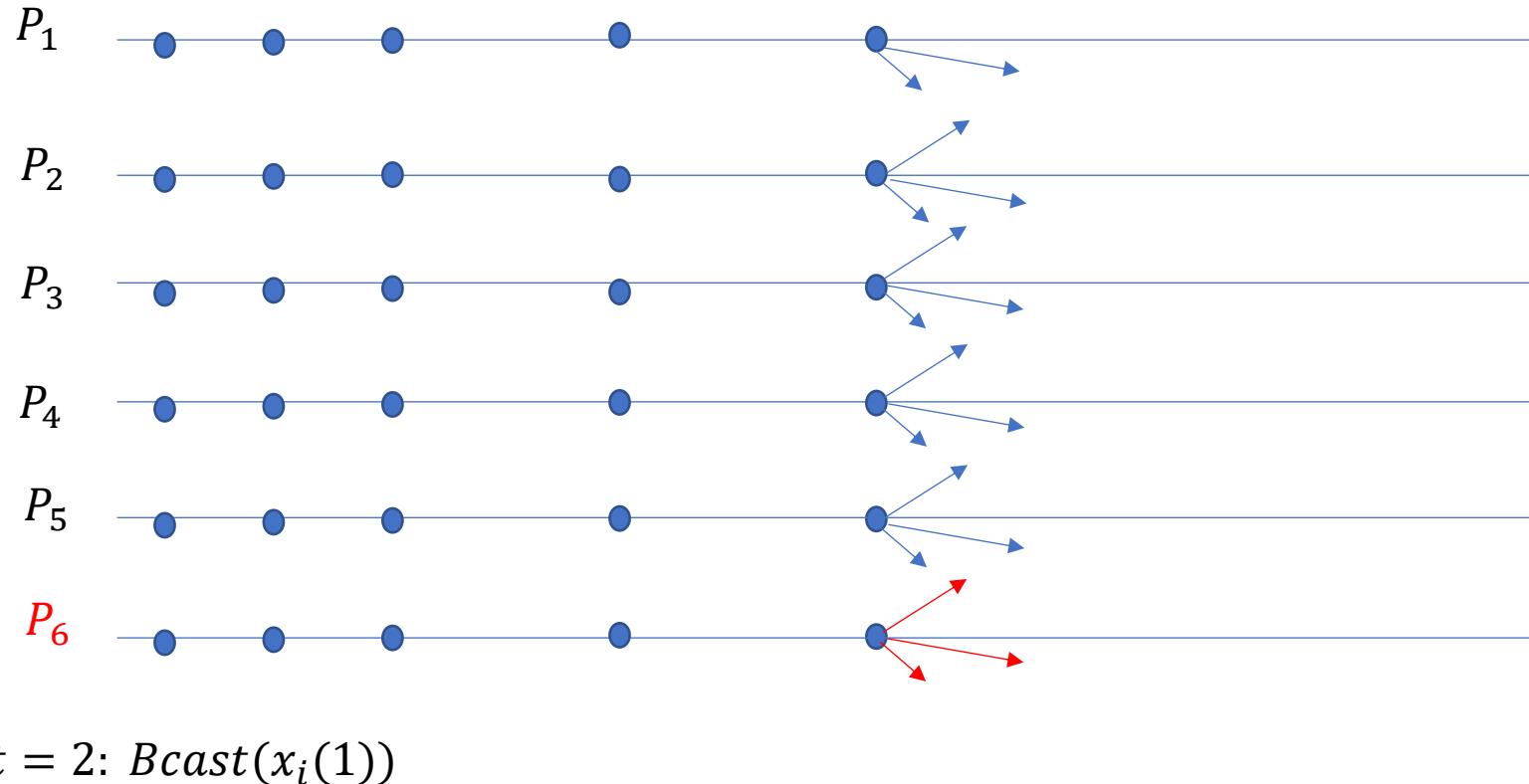
$t = 1: King \rightarrow Bcast(majority)$

Exemplu ByzFlood

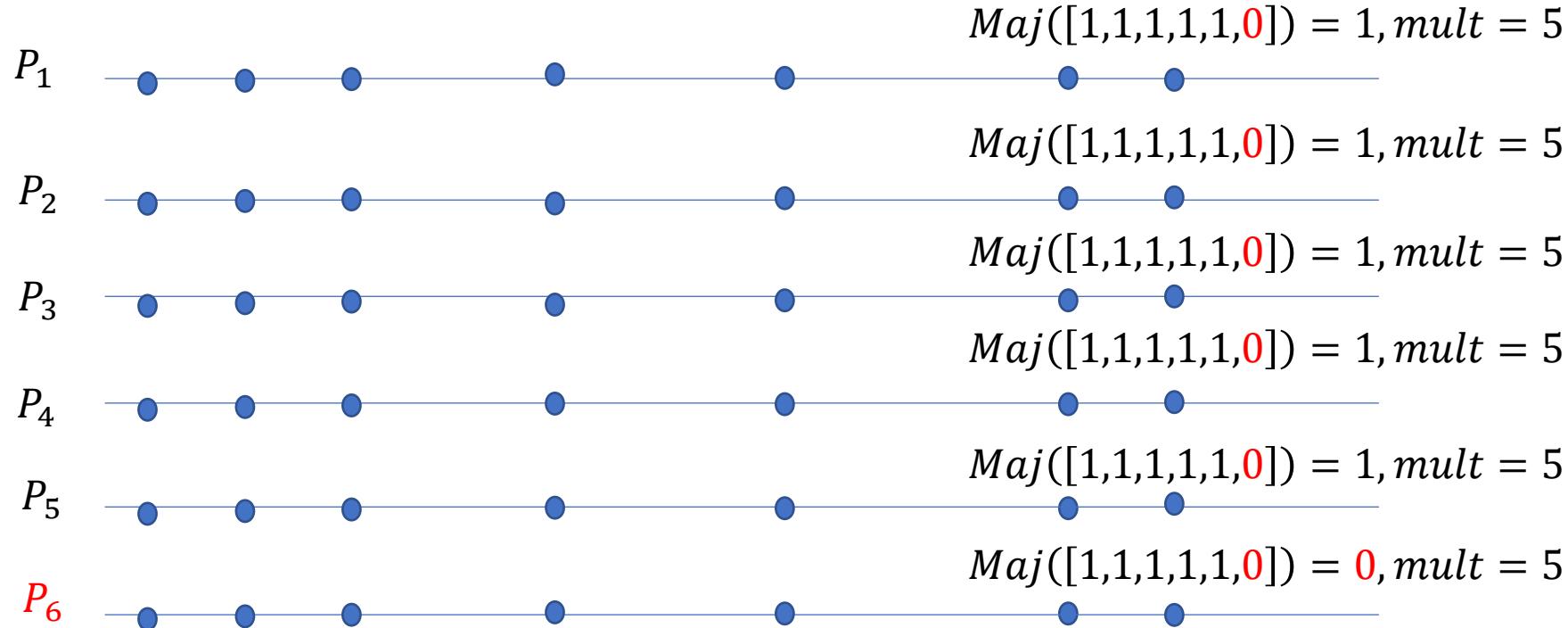


$$t = 1: \text{mult} < \frac{n}{2} + s = 4 \Rightarrow \text{majority} = \text{king} - \text{tie}$$

Exemplu ByzFlood

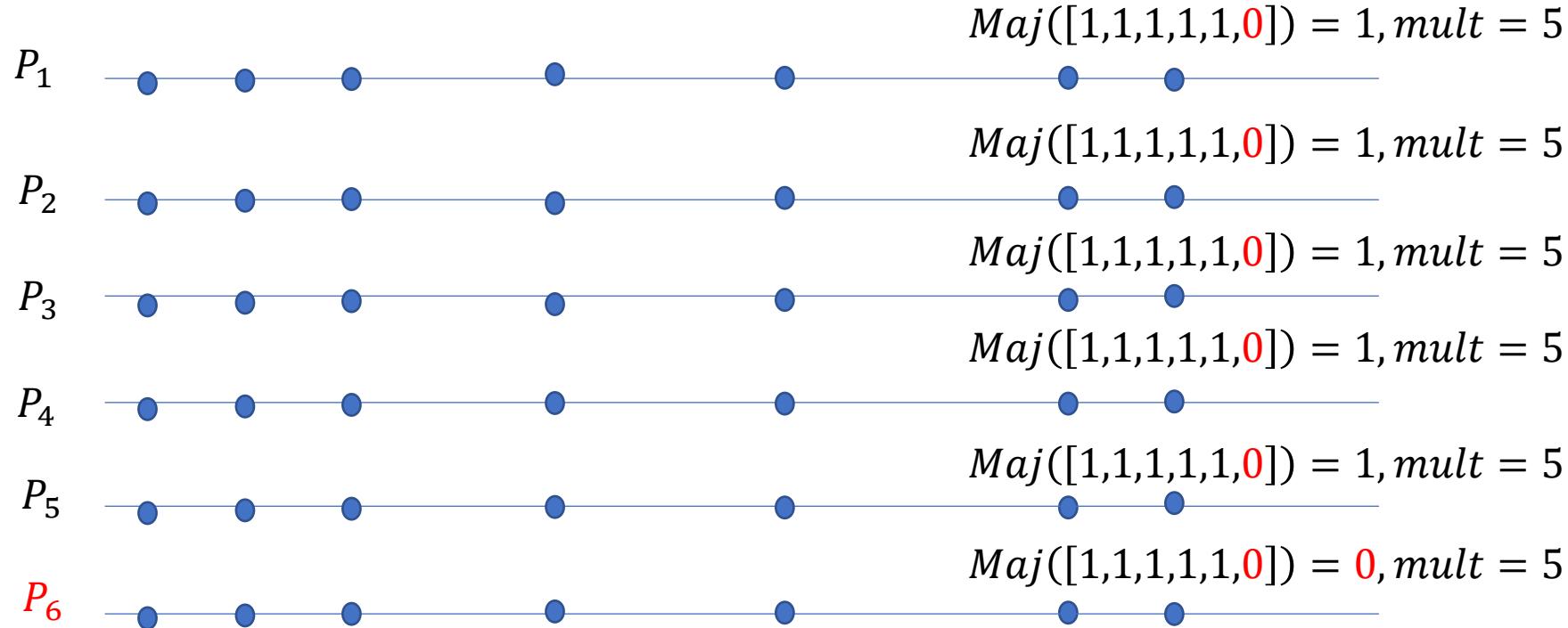


Exemplu ByzFlood



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

Exemplu ByzFlood



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

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. $mult(t) = \text{numărul de apariții al valorii } Majority(t)$

% 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 convergenței:

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 sunt primit cel puțin $n/2$ voturi pentru aceeași valoare.
3. Dacă s-a atins consensul la iterația k , atunci va persista și în iterațiile 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 - traекторie

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 traекторie/execuție este un sir (in)finit

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

Ordonarea evenimentelor

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

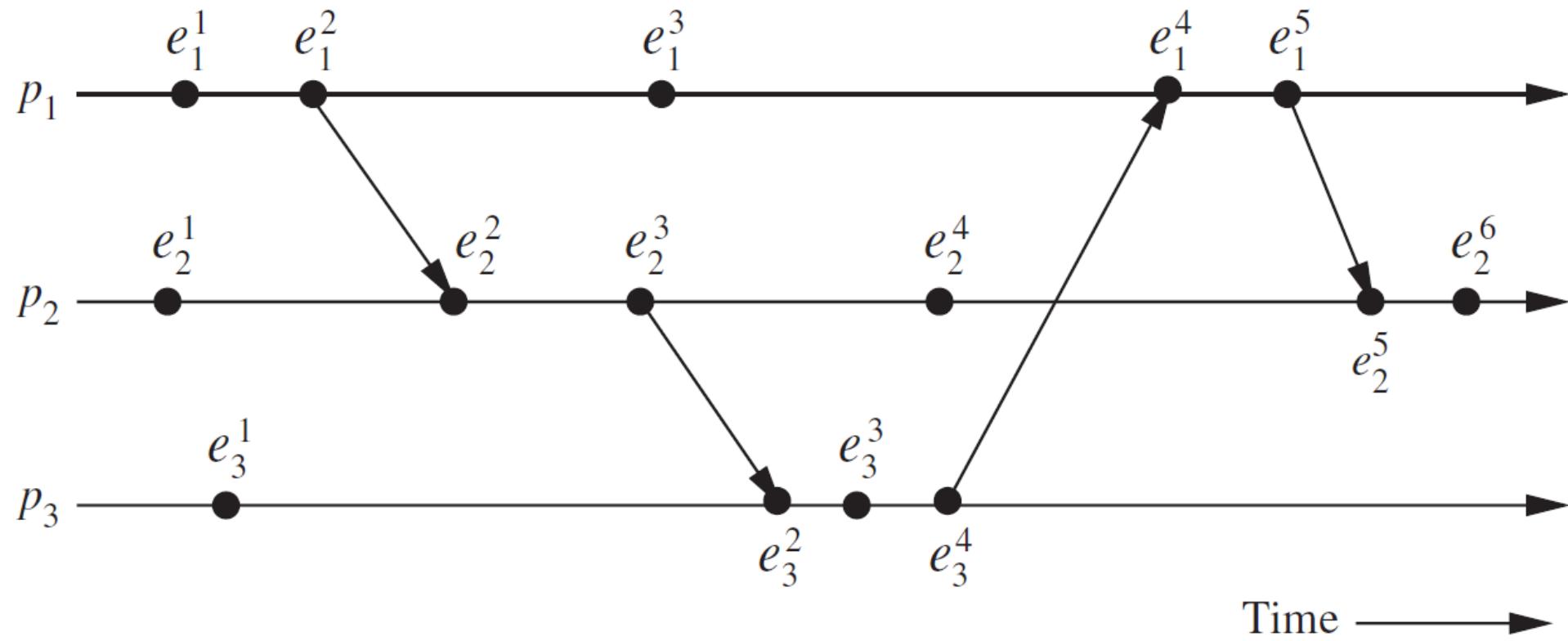
Cauzalitatea reprezintă relația dintre două (sau mai multe) evenimente în care unul are o posibilă influență asupra celorlalte.

Un eveniment e^1 (locaționalizat 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 (locaționalizat 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



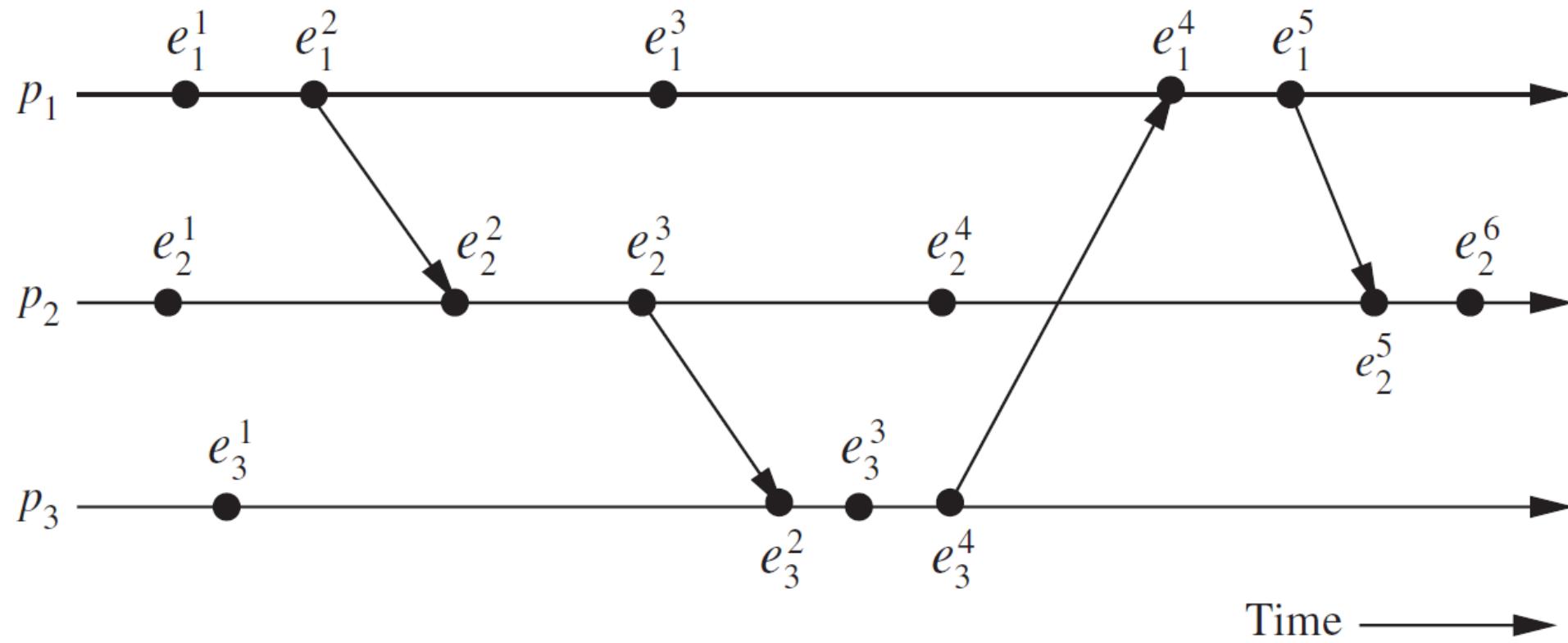
Ordine cauzală

Relația „întâmplat înainte” („*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^1 este 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 *concurrente* $e^1 || e^2$ dacă nici $e^1 \rightarrow e^2$, nici $e^2 \rightarrow e^1$ nu au loc.

Ordine cauzală



Ordine cauzală

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

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

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

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

Ordine cauzală

Problemă: Cum detectăm cauzalitatea locală a evenimentelor?

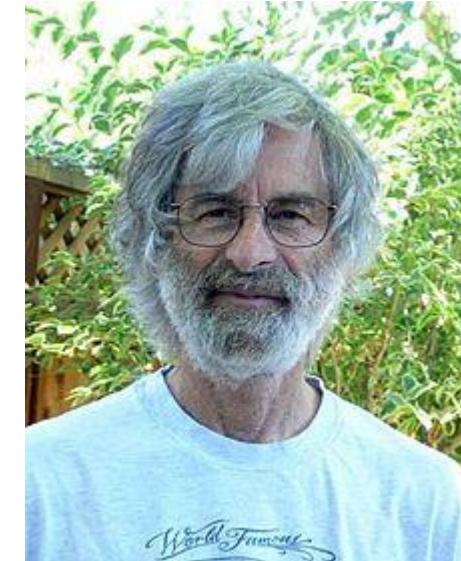
- Evenimentele locale ale unui singur proces sunt ordonate local.

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 marcat 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 perceptia 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: $C_j := \max\{C_j, C_{msg}\}$, $C_j := C_j + 1$

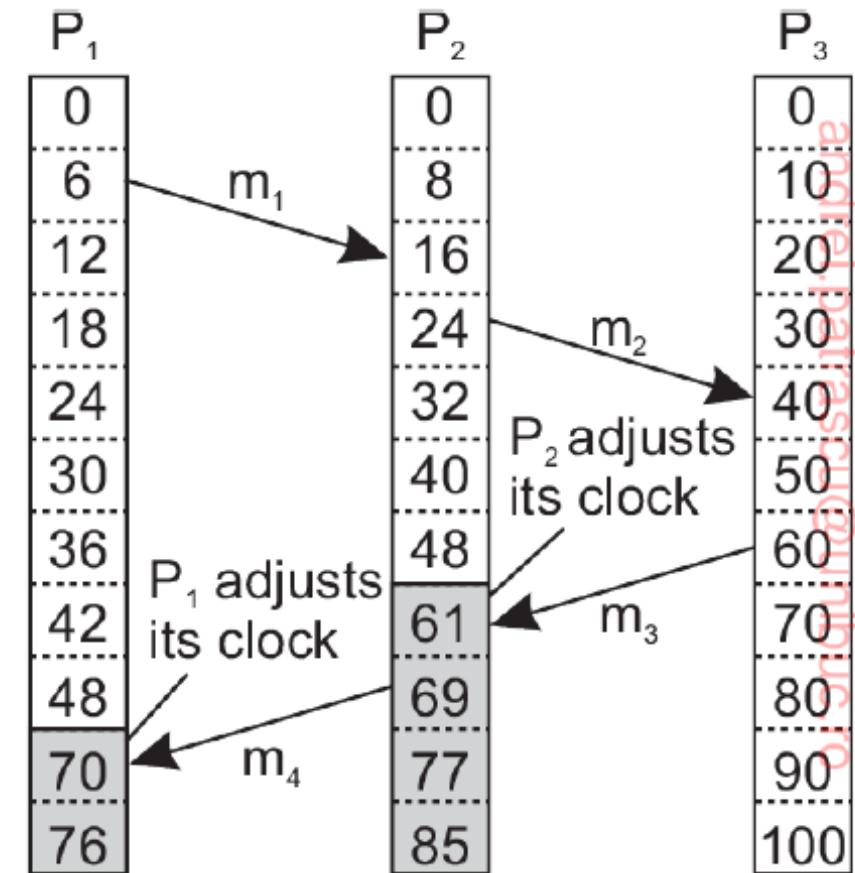
Ceasuri logice Lamport

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:

$A \rightarrow B$ implică $C(A) < C(B)$



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:

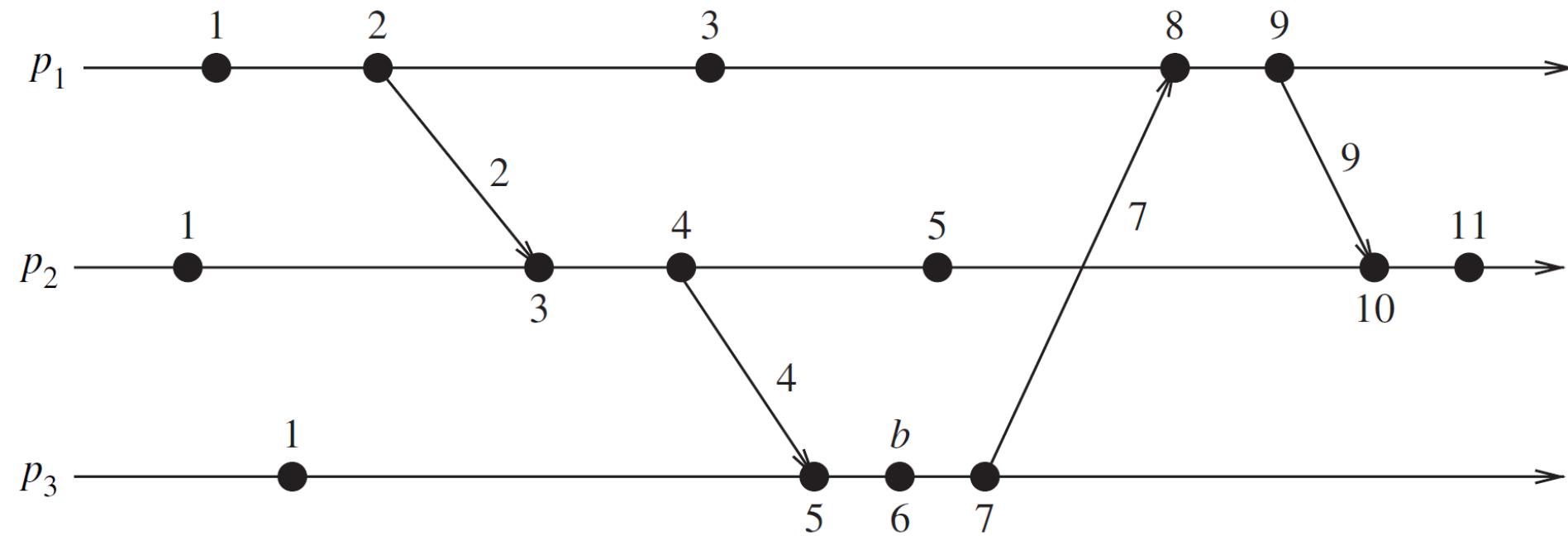
$$C_j = \max\{C_j, ts(m)\}$$

și incrementează C_j .

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

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

Ceasuri logice Lamport

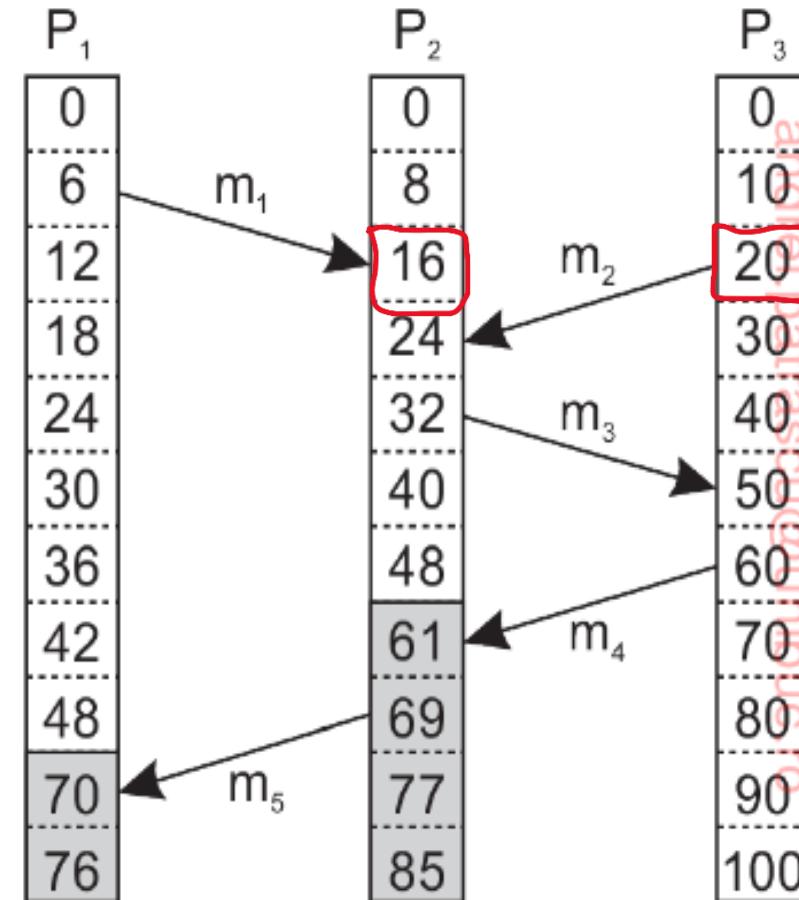


Ceasuri vectoriale

- 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 (initializat 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 : $V_i[i] = V_i[i] + 1$

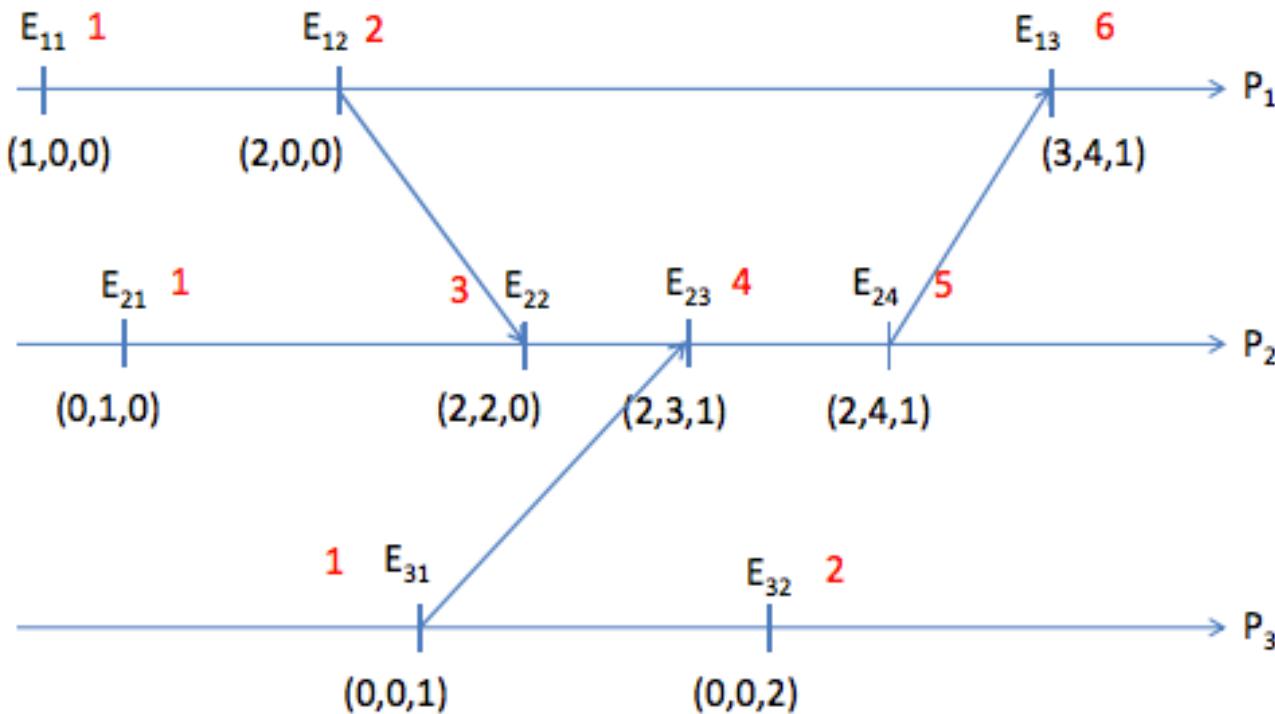
Când m este livrat de P_i la P_j atașează V_i la mesajul m

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

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

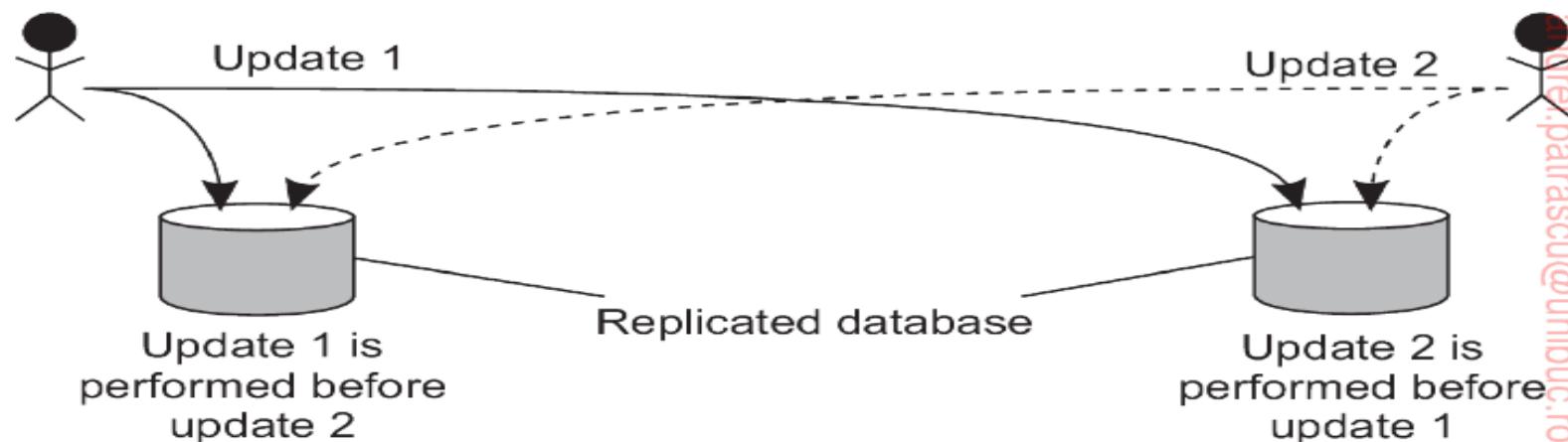
Ceasuri vectoriale

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.î. $V(A)[k] < V(B)[k]$
3. A și B sunt concurente dacă și numai dacă $V(A)! < V(B)$ și $V(B)! < V(A)$



Aplicații

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



Consistență cauzală

Bucuresti

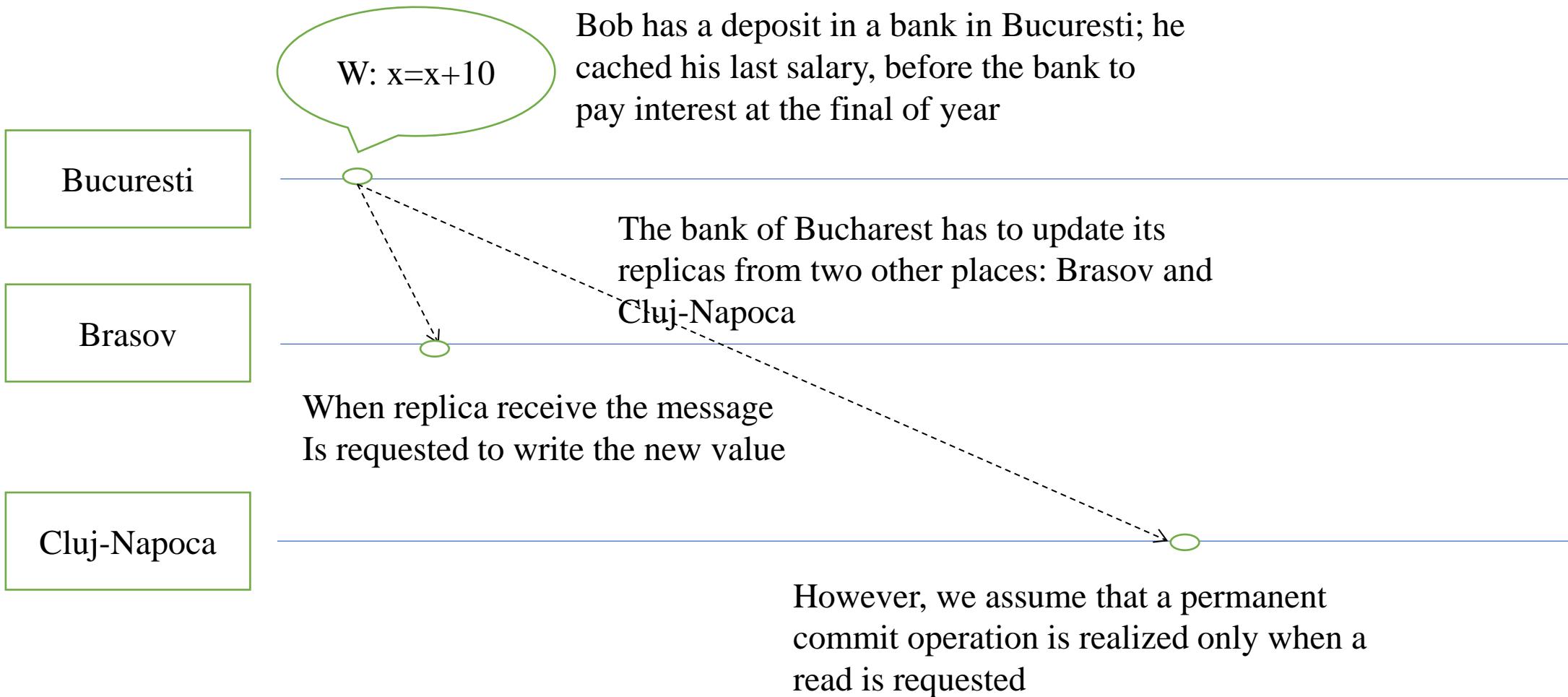
Brasov

Cluj-Napoca

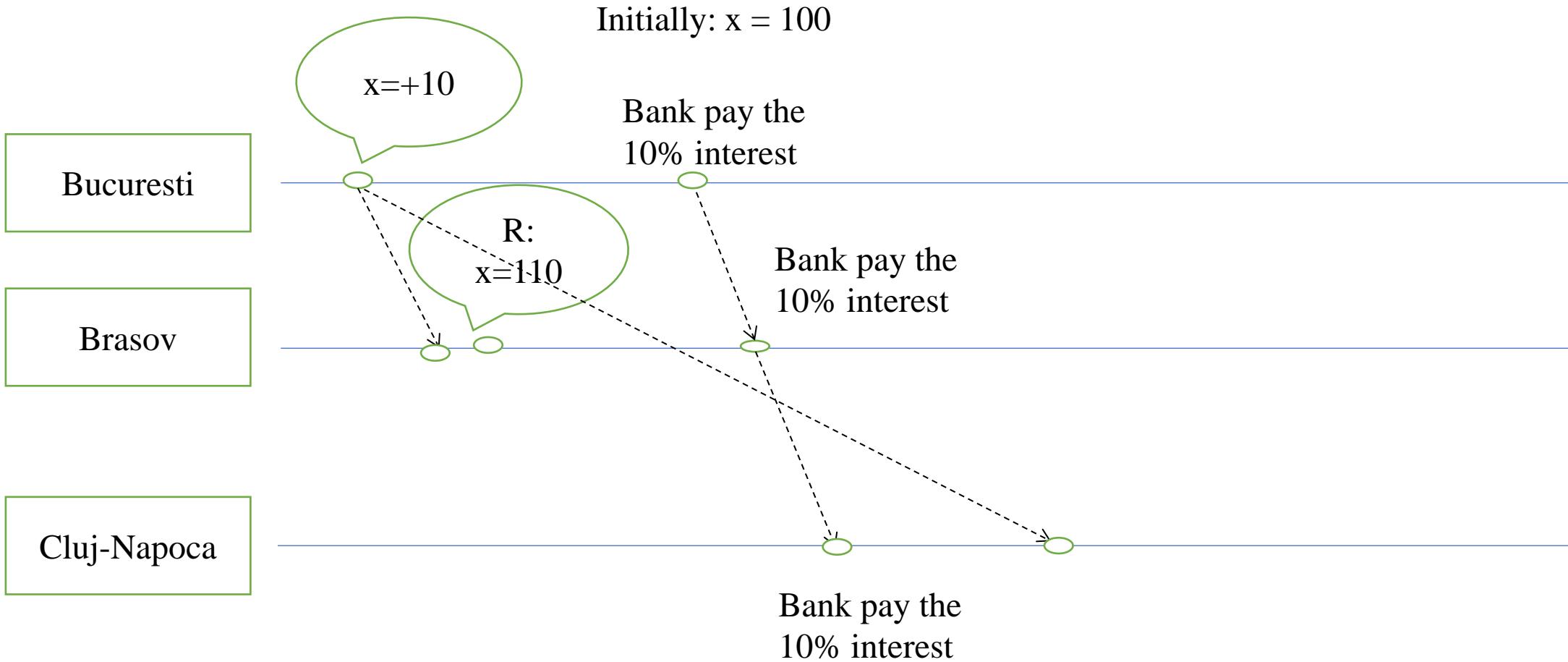
W: $x=x+10$

Bob has a deposit in a bank in Bucharest;
he cached his last salary, before the bank to
pay 10% interest at the final of year;

Consistență cauzală



Consistență cauzală



Consistență cauzală

Bucuresti:
 $x = 100$

Brasov:
 $x = 100$

Cluj-Napoca:
 $x = 100$

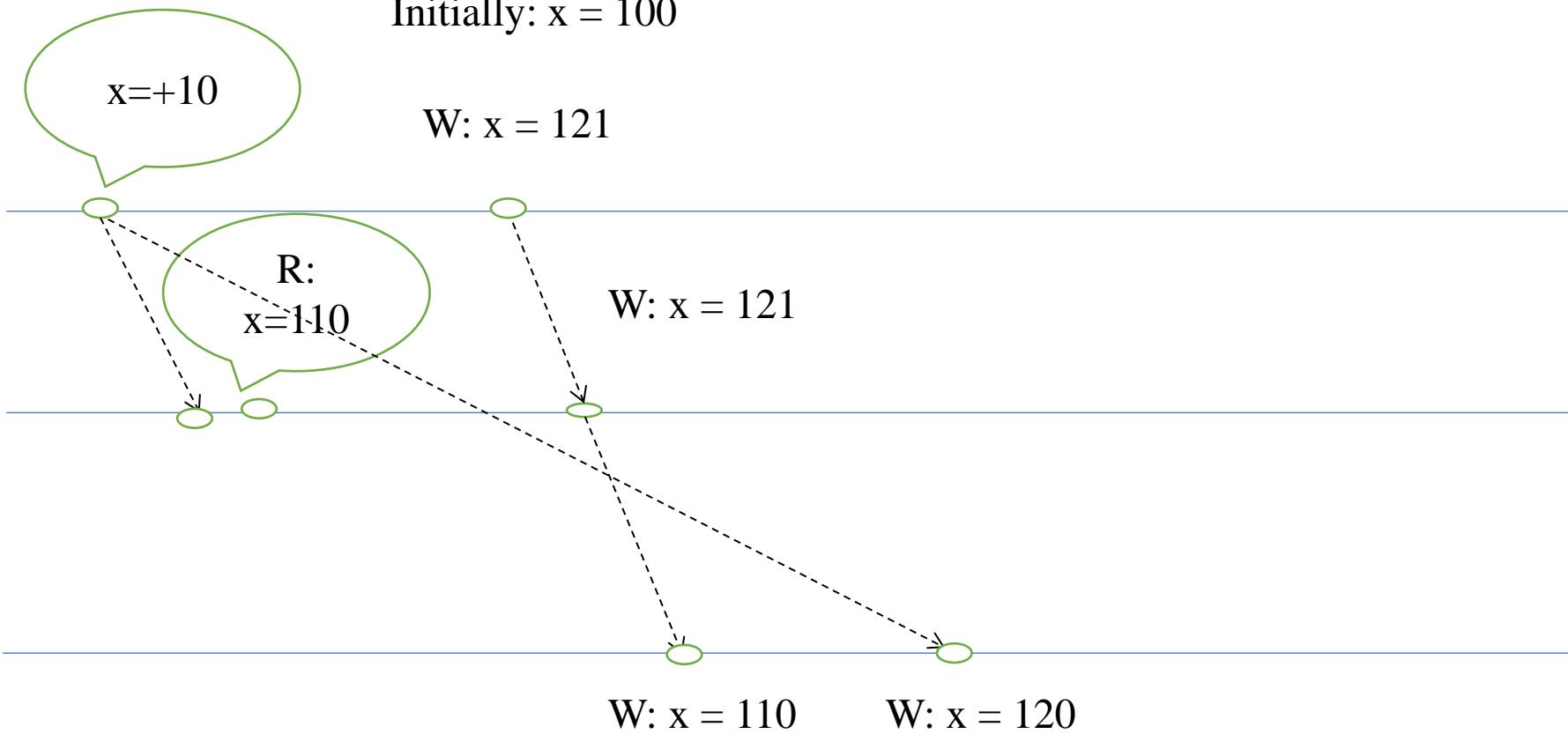
Initially: $x = 100$

$W: x = 121$

$W: x = 121$

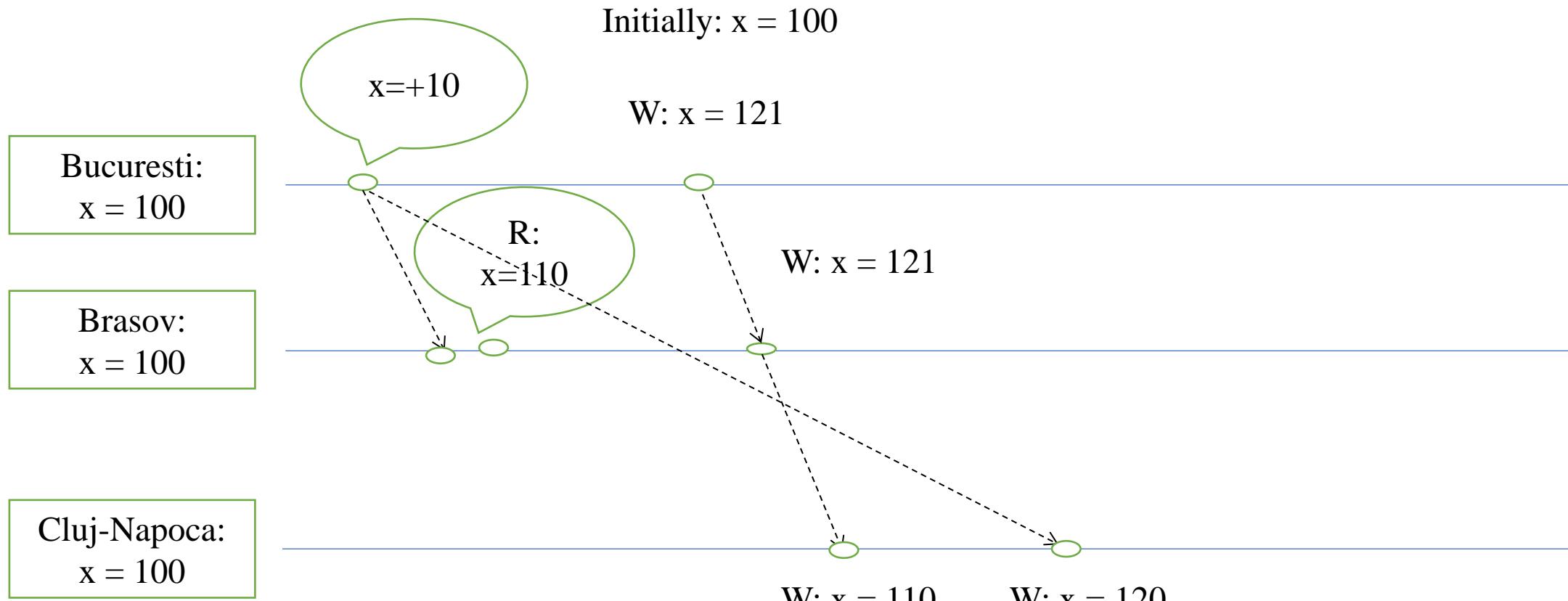
$W: x = 110$

$W: x = 120$



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

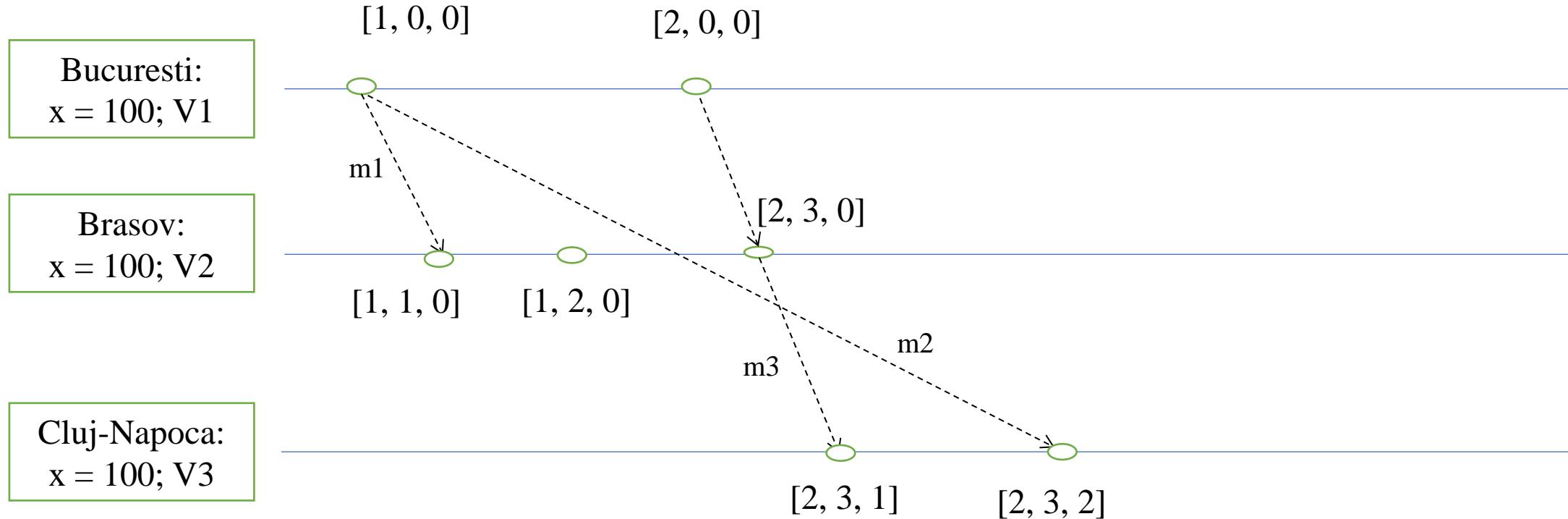
Consistență cauzală



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

Initially: $x = 100$

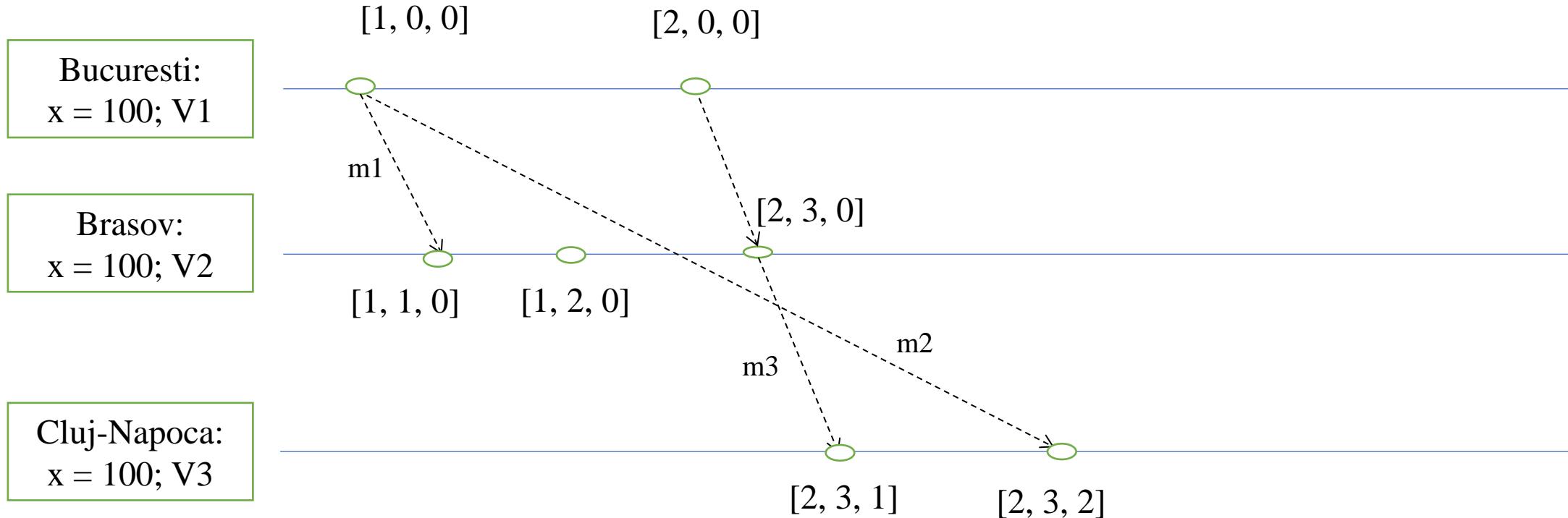


Timestamps: $ts(m1) = [1, 0, 0]; ts(m2) = [1, 0, 0]; ts(m3) = [2, 3, 0]$

Vector clocks: $V1(\text{send}(m2)) < V2(\text{send}(m3))$

Consistență cauzală

Initially: $x = 100$



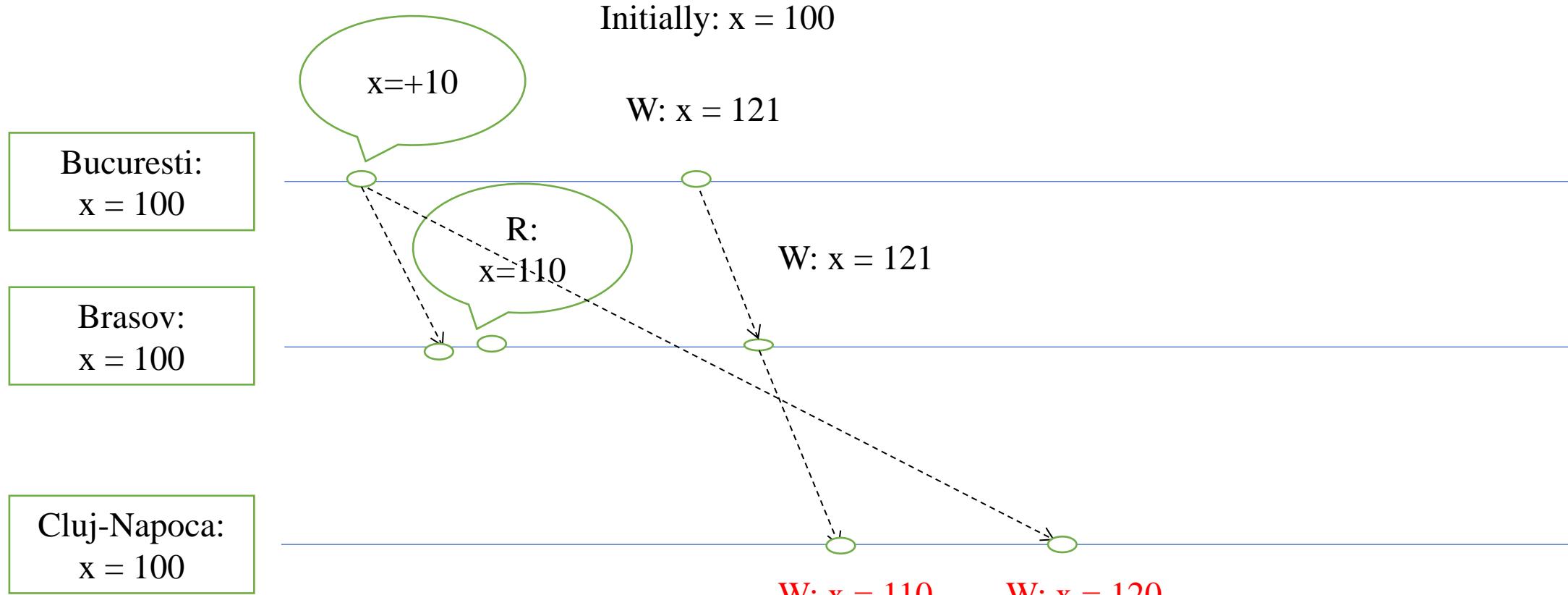
Timestamps: $ts(m1) = [1, 0, 0]$; $ts(m2) = [1, 0, 0]$; $ts(m3) = [2, 3, 0]$

Vector clocks: $V1(\text{send}(m2)) < V2(\text{send}(m3))$, 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.

Sisteme și algoritmi distribuiți

Curs 8

Recapitulare: Ceasuri vectoriale

Fiecare proces P_i stochează vectorul V_i de dimensiune n (initializat 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 : $V_i[i] = V_i[i] + 1$

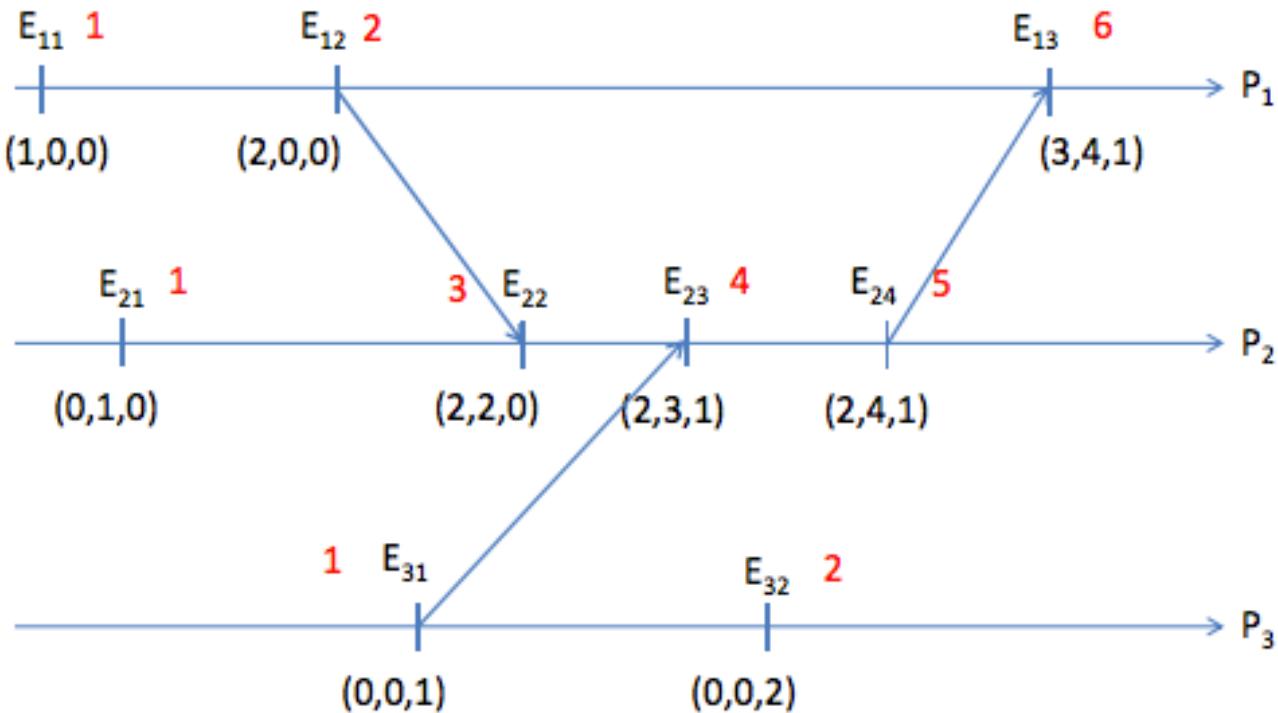
Când m este livrat de P_i la P_j atașează V_i la mesajul m

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

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.î. $V(A)[k] < V(B)[k]$
3. A și B sunt concurente dacă și numai dacă $V(A)! < V(B)$ și $V(B)! < V(A)$



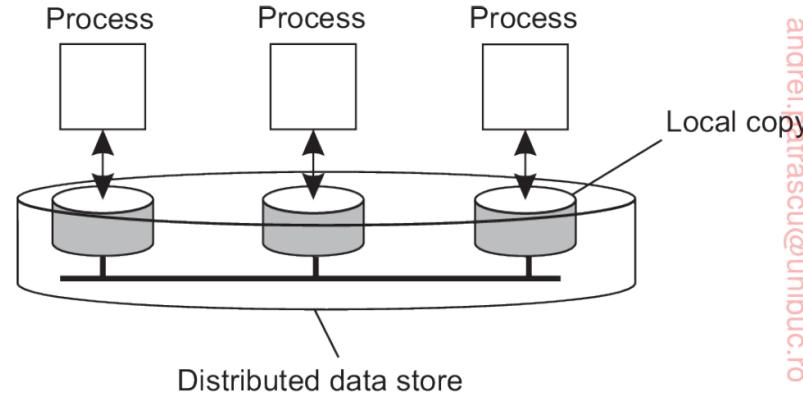
Cuprins

- 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.



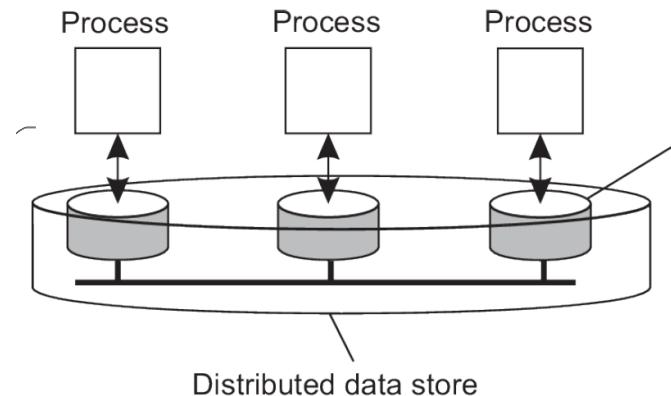
Ipoteze:

- 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 replice.

Consistență: 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, sevele 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
 - Scrimerile se execută mai întâi local și, după aceea, sunt propagate către celelalte replici.



Modele de consistență

În SDMP poate apărea inconsistență:

- 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 (protocole specifice), atunci rezultate de consistență vor fi obținute.

Modele de consistență

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

- Consistență strictă
- Consistență secvențială
- Consistență cauzală
- Consistență eventuală

Consistență 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 să dețină:

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

Consistență strictă (linearizabilitate)

$$\frac{\text{P1: } W(x)a}{\text{P2: } \quad \quad \quad R(x)a}$$

(a)

$$\frac{\text{P1: } W(x)a}{\text{P2: } \quad \quad \quad R(x)\text{NIL} \quad \quad R(x)a}$$

(b)

- a) Respectă coerență strictă
- b) Nu respectă coerență strictă

Imposibil de implementat într-un SDMP real

Consistență secvențială

Model de consistență mai relaxat decât consistență 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

Consistență secvențială

P1:	$W(x)a$	
P2:	$W(x)b$	
P3:	$R(x)b$	$R(x)a$
P4:	$R(x)b$	$R(x)a$

(a)

P1:	$W(x)a$	
P2:	$W(x)b$	
P3:	$R(x)b$	$R(x)a$
P4:	$R(x)a$	$R(x)b$

(b)

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

Consistență secvențială

Process P ₁	Process P ₂	Process P ₃
$x \leftarrow 1;$ <code>print(y, z);</code>	$y \leftarrow 1;$ <code>print(x, z);</code>	$z \leftarrow 1;$ <code>print(x, y);</code>

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

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

Consistență secvențială

Process P ₁	Process P ₂	Process P ₃
$x \leftarrow 1;$ <code>print(y,z);</code>	$y \leftarrow 1;$ <code>print(x,z);</code>	$z \leftarrow 1;$ <code>print(x,y);</code>

Execution 1	Execution 2	Execution 3	Execution 4
P ₁ : $x \leftarrow 1;$ P ₁ : <code>print(y,z);</code>	P ₁ : $x \leftarrow 1;$ P ₂ : $y \leftarrow 1;$ P ₂ : <code>print(x,z);</code>	P ₂ : $y \leftarrow 1;$ P ₃ : $z \leftarrow 1;$ P ₃ : <code>print(x,y);</code>	P ₂ : $y \leftarrow 1;$ P ₁ : $x \leftarrow 1;$ P ₃ : $z \leftarrow 1;$ P ₂ : <code>print(x,z);</code>
P ₂ : $y \leftarrow 1;$ P ₂ : <code>print(x,z);</code>	P ₂ : $y \leftarrow 1;$ P ₁ : <code>print(y,z);</code>	P ₃ : $z \leftarrow 1;$ P ₃ : <code>print(x,y);</code>	P ₁ : $x \leftarrow 1;$ P ₁ : <code>print(y,z);</code>
P ₃ : $z \leftarrow 1;$ P ₃ : <code>print(x,y);</code>	P ₃ : $z \leftarrow 1;$ P ₃ : <code>print(x,y);</code>	P ₁ : $x \leftarrow 1;$ P ₁ : <code>print(y,z);</code>	P ₃ : $z \leftarrow 1;$ P ₃ : <code>print(x,y);</code>
<i>Prints: 001011</i> <i>Signature: 00 10 11</i>	<i>Prints: 101011</i> <i>Signature: 10 10 11</i>	<i>Prints: 010111</i> <i>Signature: 11 01 01</i>	<i>Prints: 111111</i> <i>Signature: 11 11 11</i>

Consistență secvențială

P1:	W(x)a	W(y)a	R(x)b
<hr/>			
P2:	W(y)b	W(x)b	R(y)a

ascu

Pentru aceste operații avem CS

- Dacă urmărим 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).

Consistență secvențială

$$\begin{array}{cccc} P1: & W(x)a & W(y)a & R(x)b \\ \hline P2: & W(y)b & W(x)b & R(y)a \end{array}$$

ascu

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

Consistență 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.

Consistență cauzală

- a) Violarea consistenței cauzale – scrierea din P1 este în relație cauzală cu scrierea din P2 și 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.

P1:	W(x)a	
P2:	R(x)a	W(x)b
P3:		R(x)b R(x)a
P4:		R(x)a R(x)b

(a)

P1:	W(x)a	
P2:		W(x)b
P3:		R(x)b R(x)a
P4:		R(x)a R(x)b

(b)

Consistență cauzală

P1:	W(x)a	
P2:	R(x)a	W(y)b
P3:		R(y)b R(x)?
P4:		R(x)a R(y)?

)atrascu@ui

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 $R_4(x)$

- Cu toate că avem formal relația $W_1(x)a \rightarrow W_2(y)b$, initializările variabilelor sunt independente.
- De aceea, $R_4(x)NULL$ se conformează consistenței cauzale.

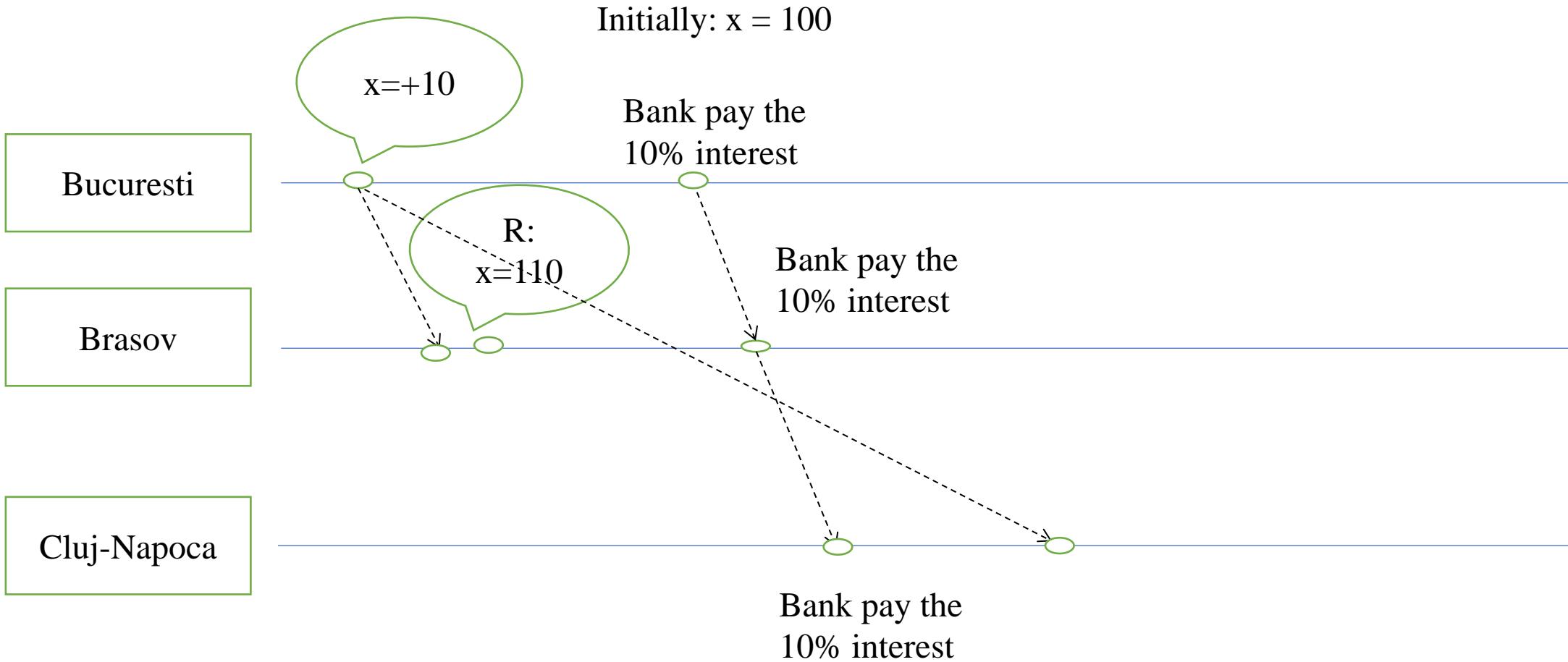
Exercițiu

Care model de consistență se respectă în următorul scenariu?

P1:	$W(x)a$		$W(x)c$	
P2:		$R(x)a$	$W(x)b$	
P3:		$R(x)a$	$R(x)c$	$R(x)b$
P4:		$R(x)a$	$R(x)b$	$R(x)c$

)atrascu@uni

Consistență cauzală



Consistență cauzală

Bucuresti:
 $x = 100$

Brasov:
 $x = 100$

Cluj-Napoca:
 $x = 100$

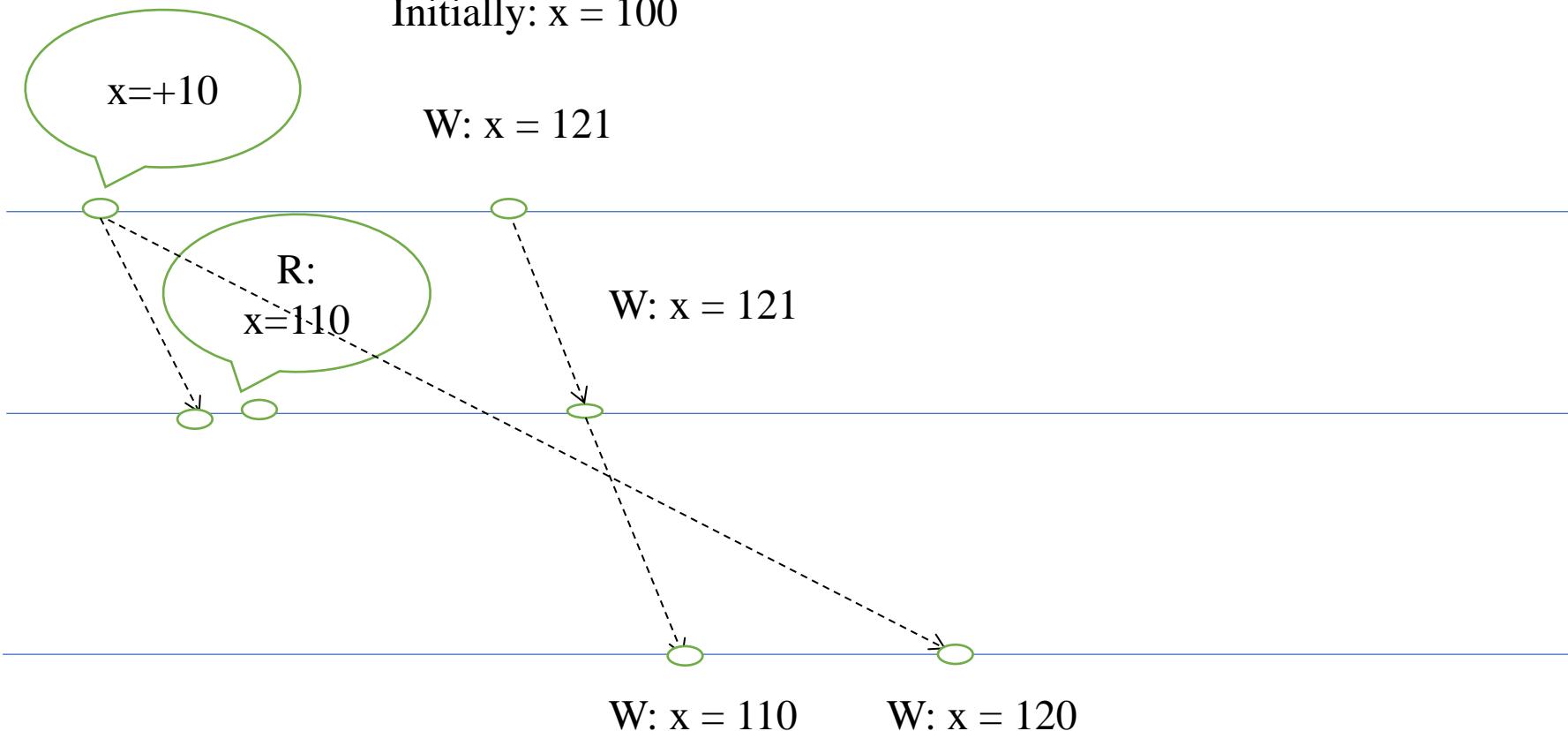
Initially: $x = 100$

$W: x = 121$

$W: x = 121$

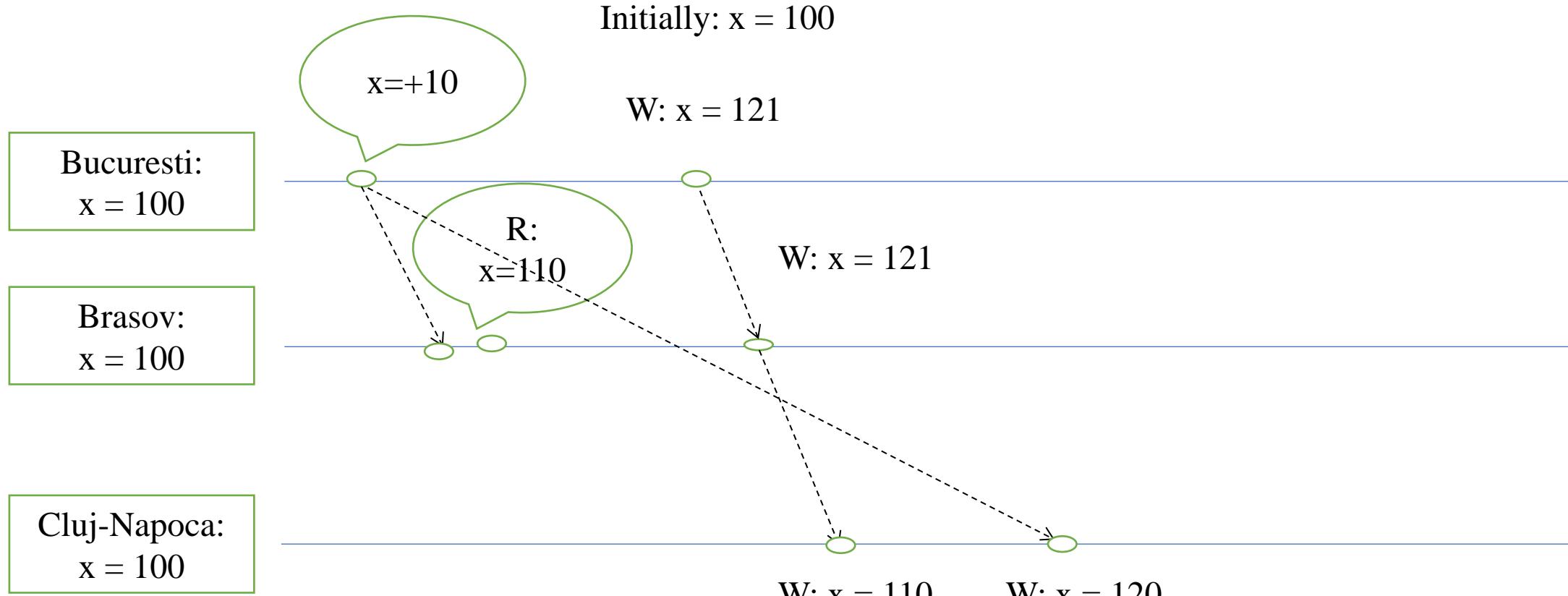
$W: x = 110$

$W: x = 120$



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

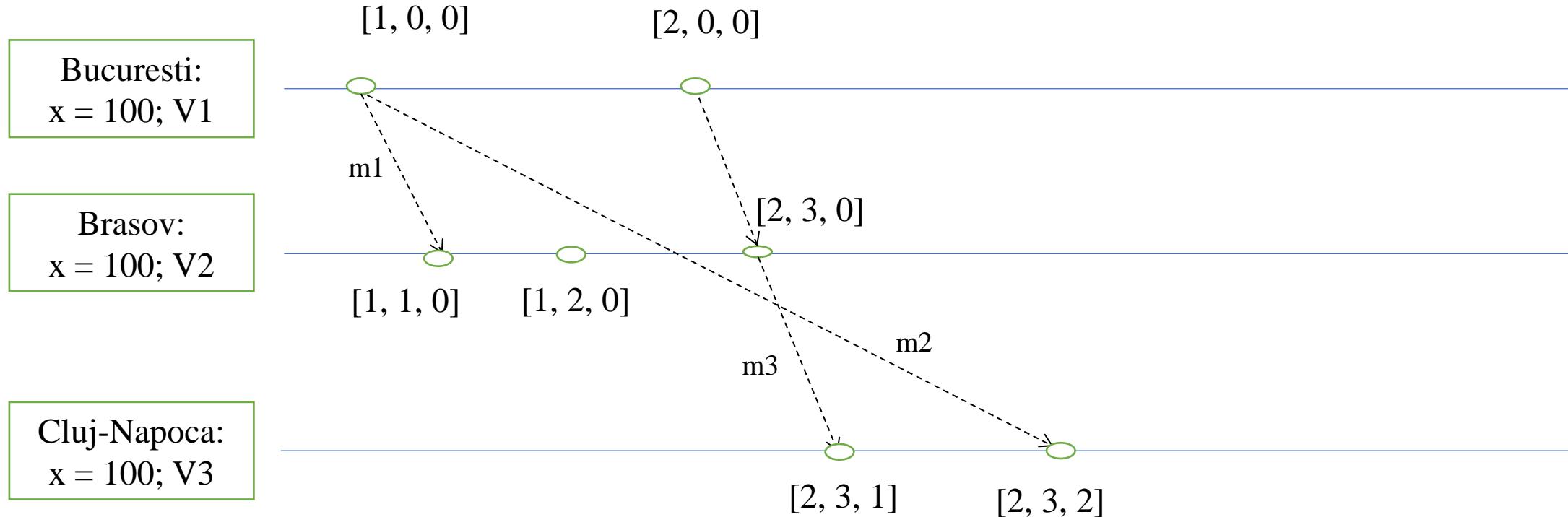
Consistență cauzală



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

Initially: $x = 100$

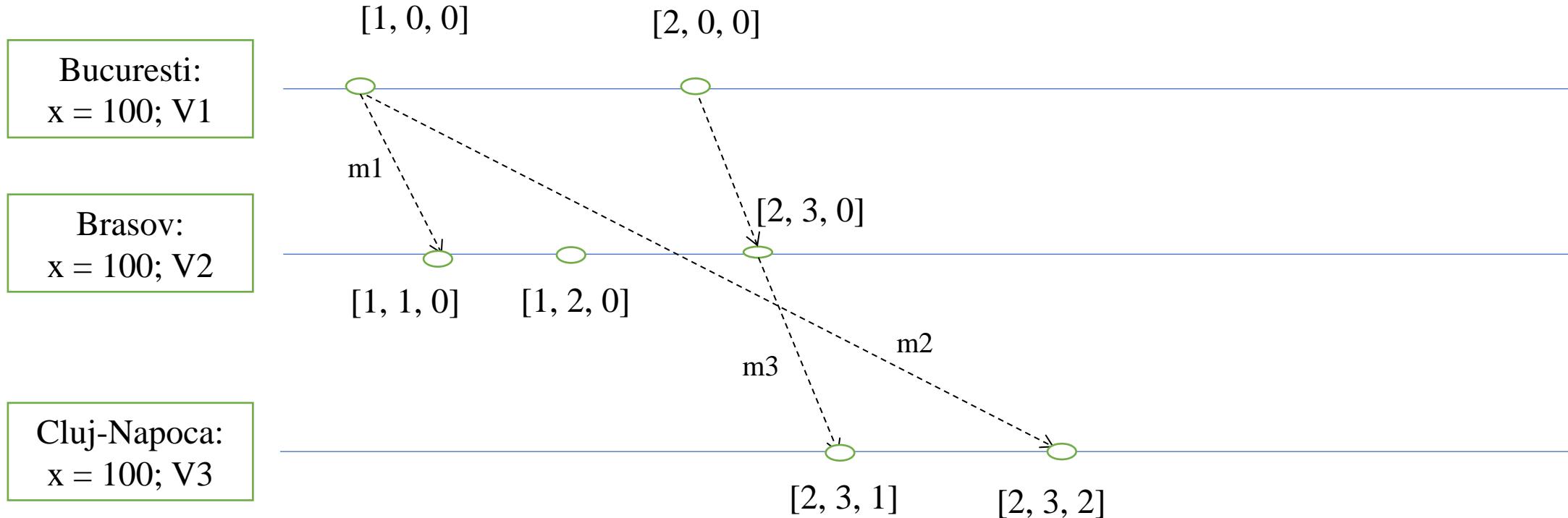


Timestamps: $ts(m1) = [1, 0, 0]; ts(m2) = [1, 0, 0]; ts(m3) = [2, 3, 0]$

Vector clocks: $V1(\text{send}(m2)) < V2(\text{send}(m3))$

Consistență cauzală

Initially: $x = 100$



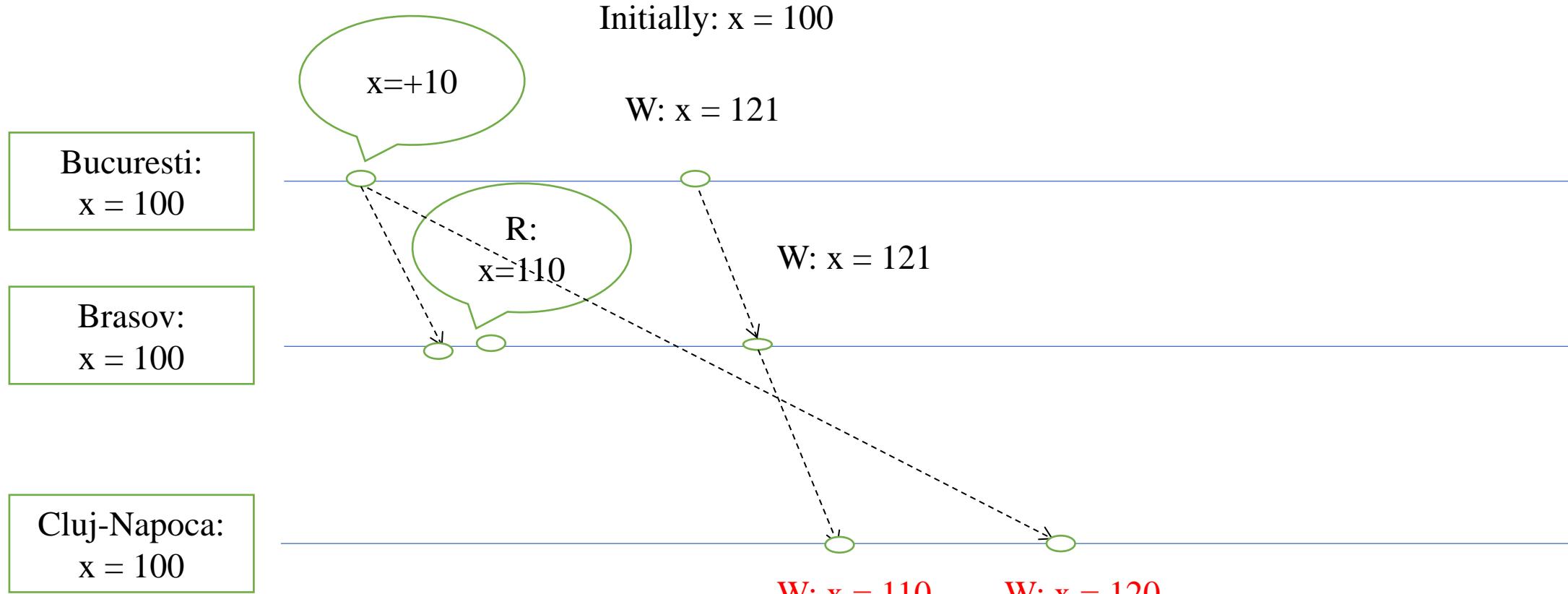
Timestamps: $ts(m1) = [1, 0, 0]$; $ts(m2) = [1, 0, 0]$; $ts(m3) = [2, 3, 0]$

Vector clocks: $V1(\text{send}(m2)) < V2(\text{send}(m3))$, 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 coherență 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

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

Algoritmi

Două scheme simple pentru a păstra consistență 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 distanță
 - 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.

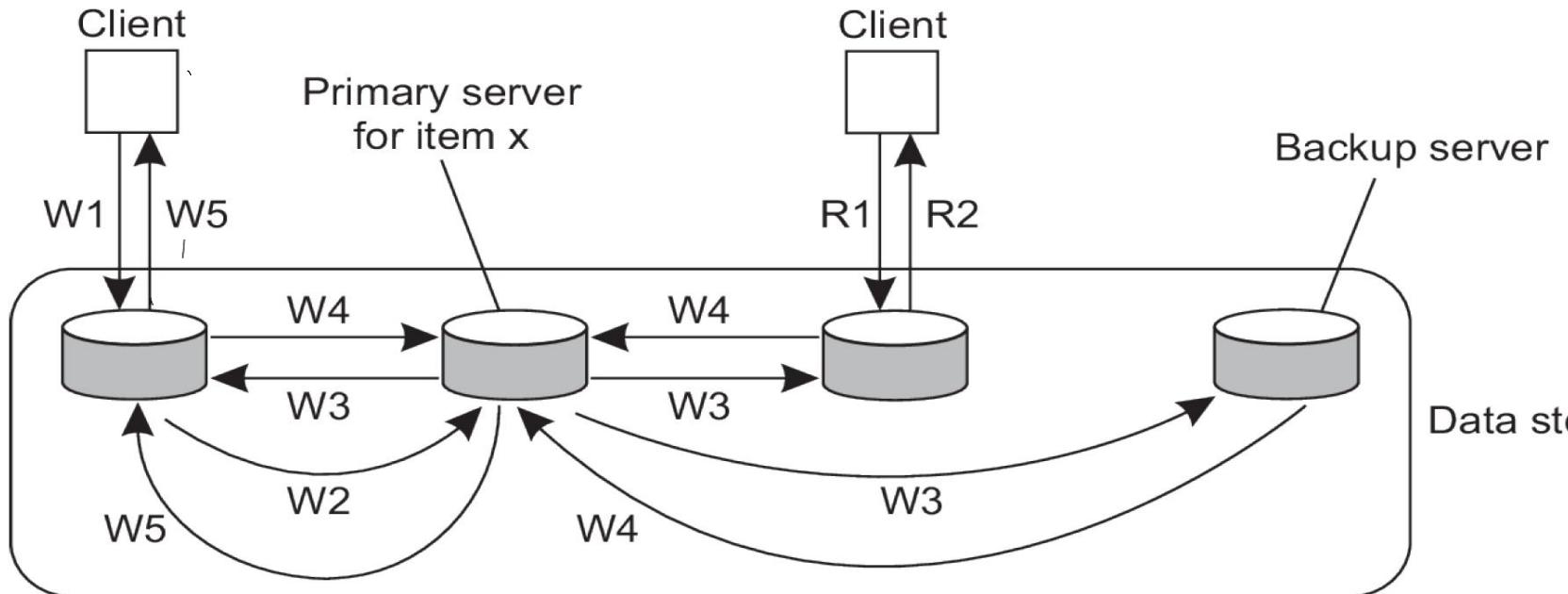
Schema Remote-write

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

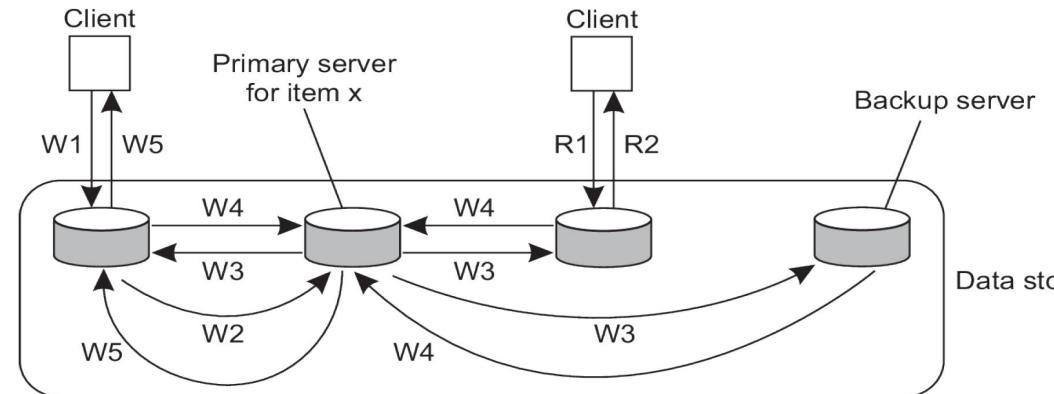
R1. Read request
R2. Response to read

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

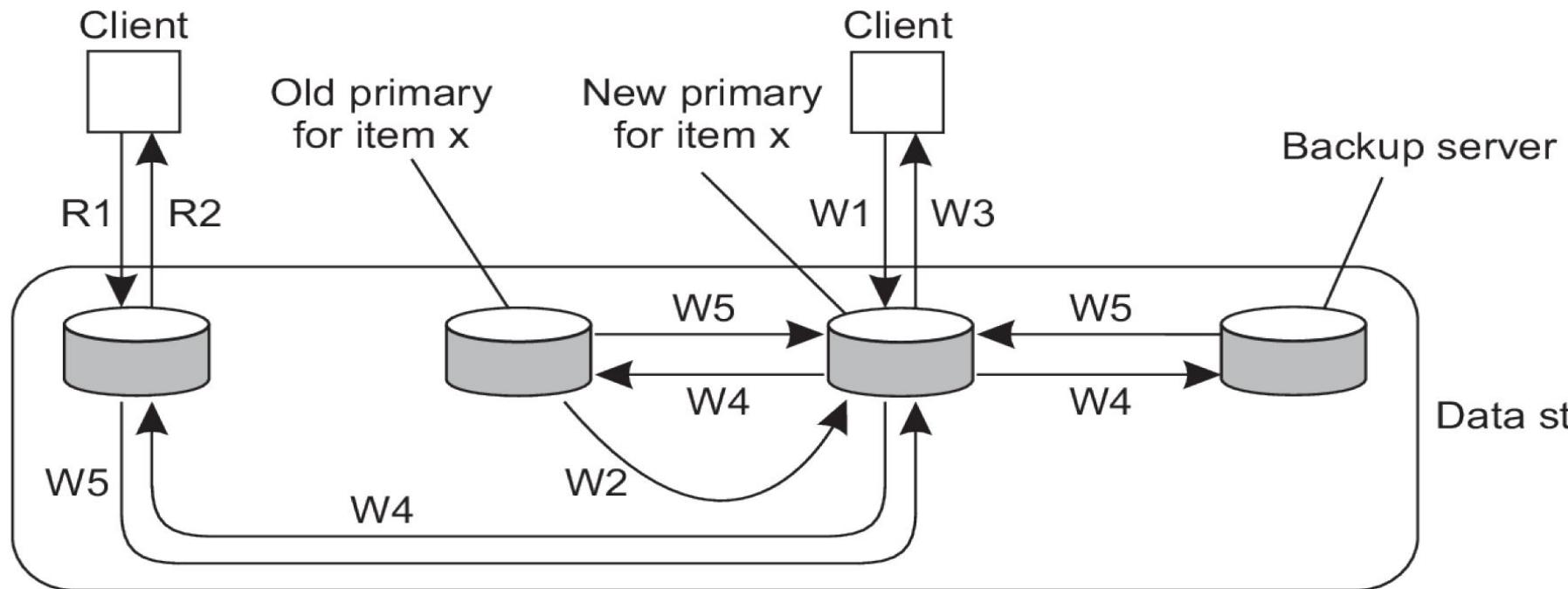
R1. Read request
R2. Response to read

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 successive 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

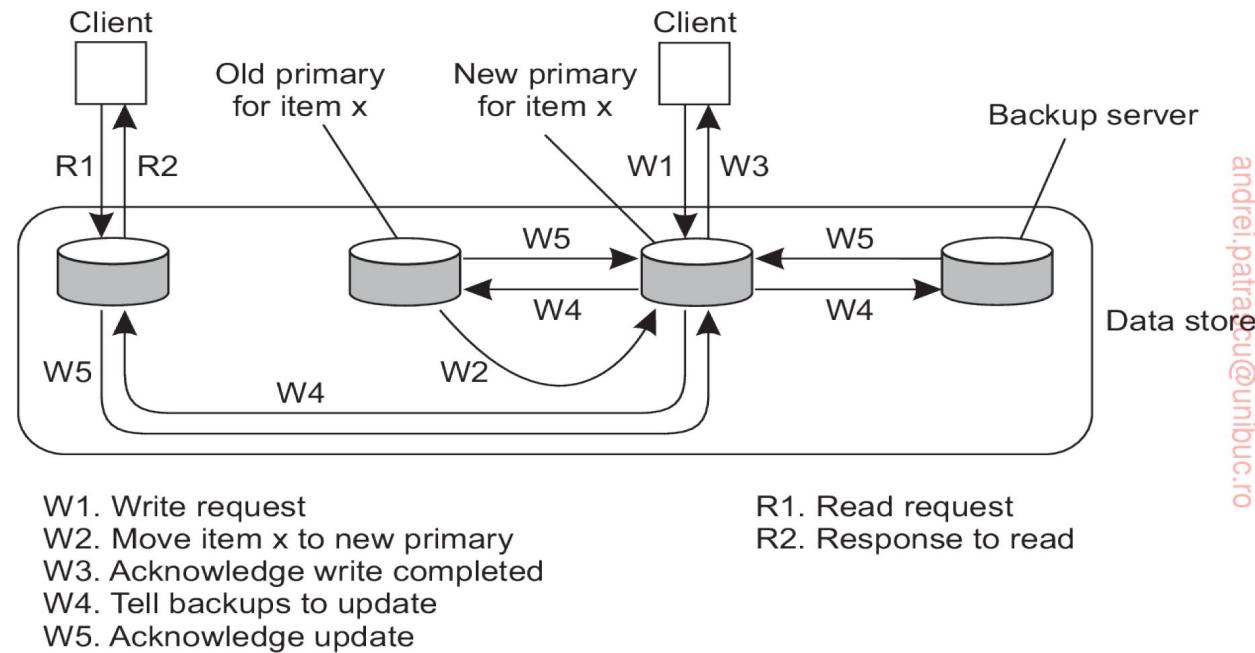
R2. Response to read

andrei.patraescu@unibuc.ro

Schema Local-write

Specific aplicațiilor de tipul *offline repository*, în care operațiile de scriere au loc local, iar după 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

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

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*

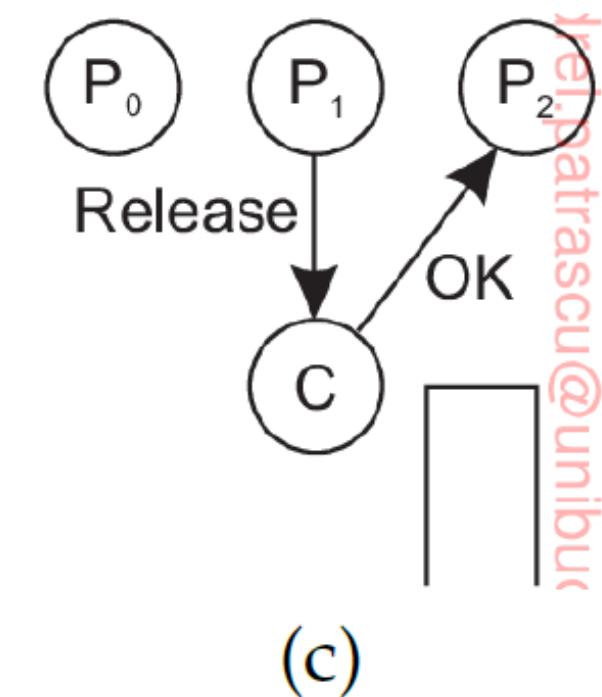
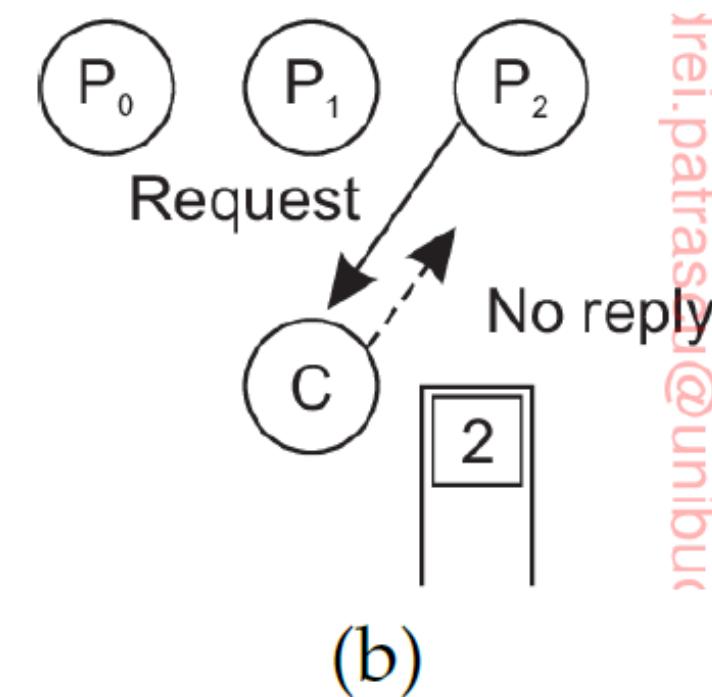
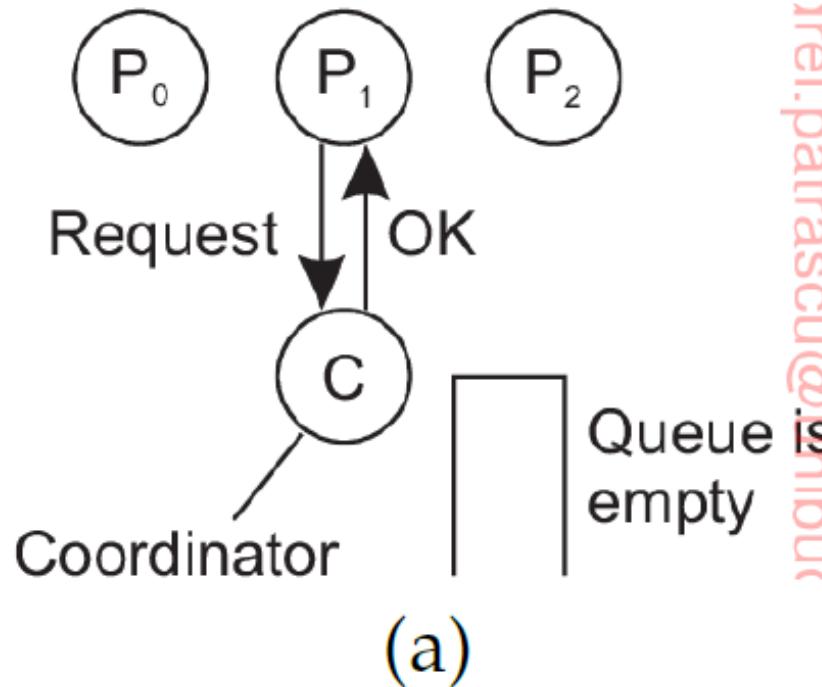
Pentru ieșire (release): send *release_message*

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



Jrei.patrascu@unibuc.ro

Jrei.patrascu@unibuc.ro

Jrei.patrascu@unibuc.ro

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 $n \rightarrow \infty$, 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 proceze.
 4. Intră în secțiunea critică.

Excludere mutuală distribuită

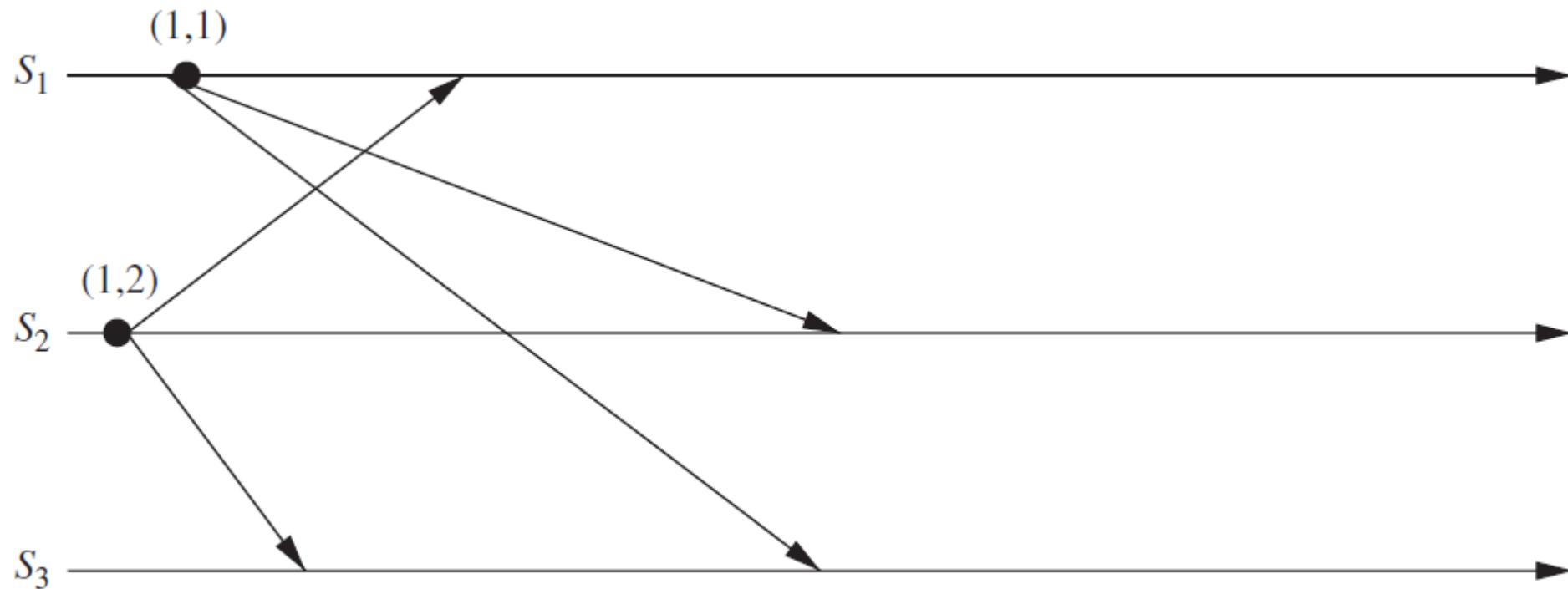
Premisă: Fiecare proces P_i păstrează un ceas logic L_i .

2. Când P_j primește un mesaj de la P_i :

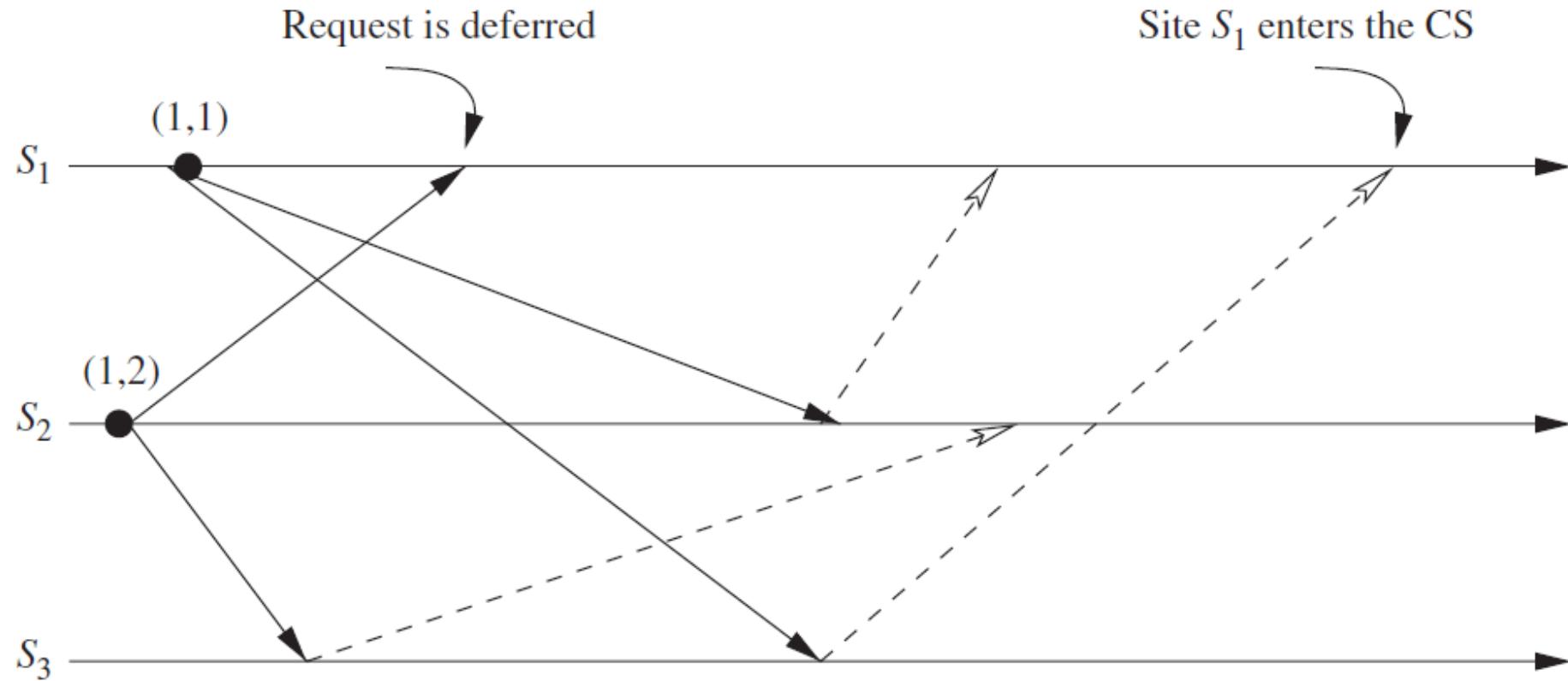
1. Dacă se află în afara secțiunii critice: *send OK*.
2. Dacă se află în secțiunea critică: nu răspunde, adaugă *request* în coadă.
3. Dacă intenționează să intre în secțiunea critică:
 1. dacă $(L_i, i) < (L_j, j)$: *send OK*
 2. altfel: adaugă *request* în coadă.

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

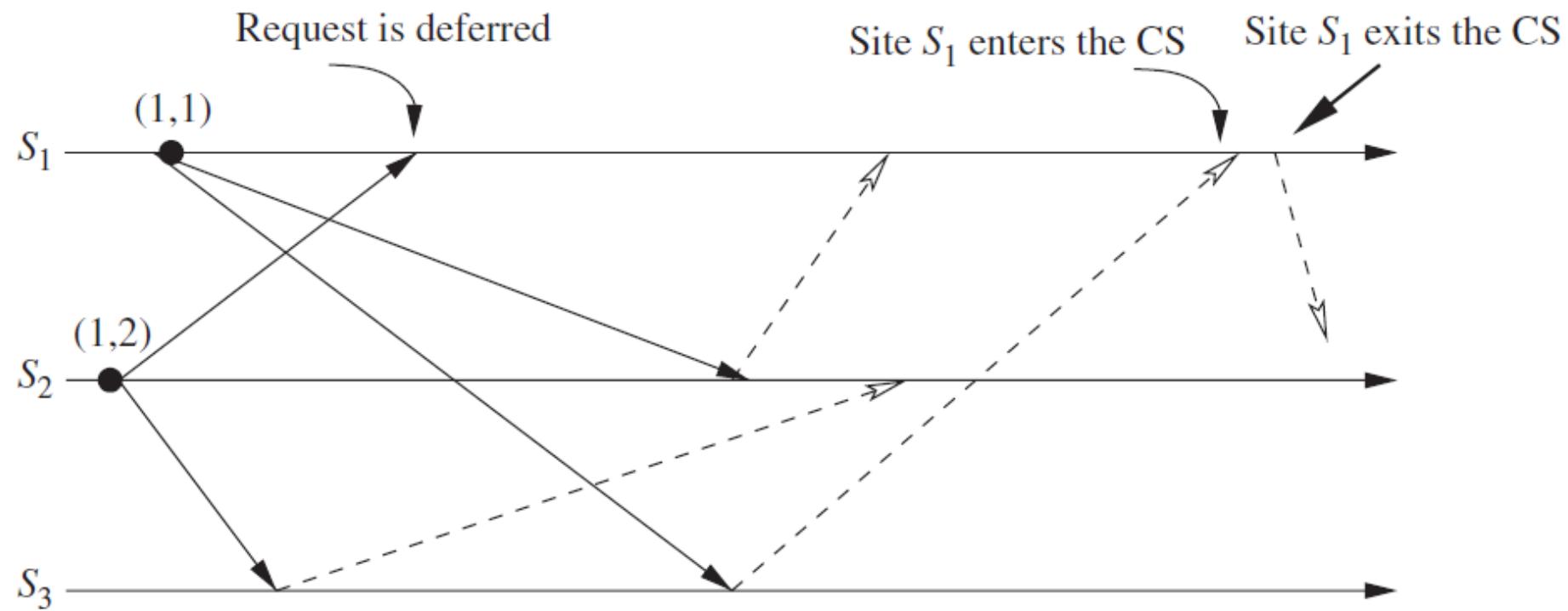
Excludere mutuală distribuită



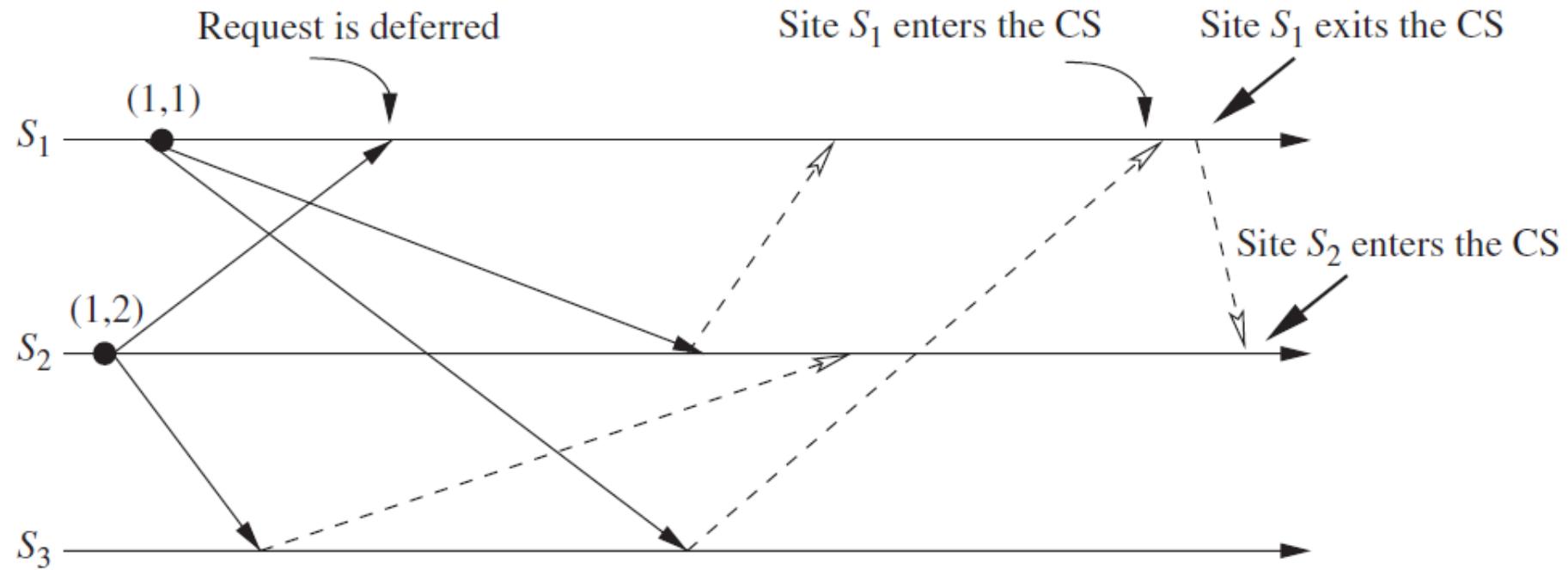
Excludere mutuală distribuită



Excludere mutuală distribuită



Excludere mutuală distribuită



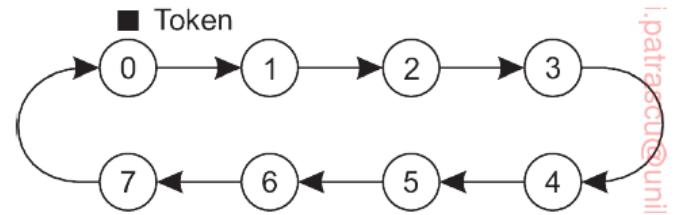
Excludere mutuală distribuită

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 intră 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!

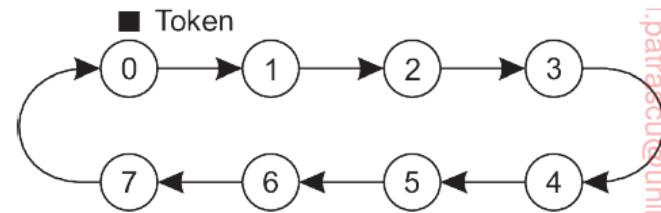


Algoritm Token Ring

- 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

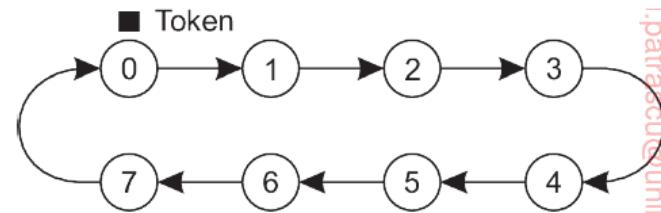
Când P_i primește jetonul:

- 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 îintrerup inelul.
- Dezavantaj: Apar diferențe de răspuns în situațiile (T durată livrare token, E timp execuție SC, n noduri):
 - Încărcarea este **slabă**
 - Încărcarea este **puternică**



i.patraescu@unil

Sisteme și algoritmi distribuiți

Curs 10

Cuprins

- Algoritmi Gossip (partea a II-a)
- Sisteme liniare. Introducere în problema intersecției de multimi convexe.
- Algoritmi distribuiți de consens pentru sisteme liniare.

Un model simplu

Schema Gossip de bază (pe perechi coordonate):

1. Algoritmul utilizează informația primită de la vecini $N(i)$
2. Realizează o operație de actualizare (simplă) per unitate de timp
- 3. Nodul i selectează aleator un vecin și trimite actualizarea** (schema *push*)
4. Eventual, nodul i realizează o operație *pull*.

În problema de medie operația de actualizare se reduce la:

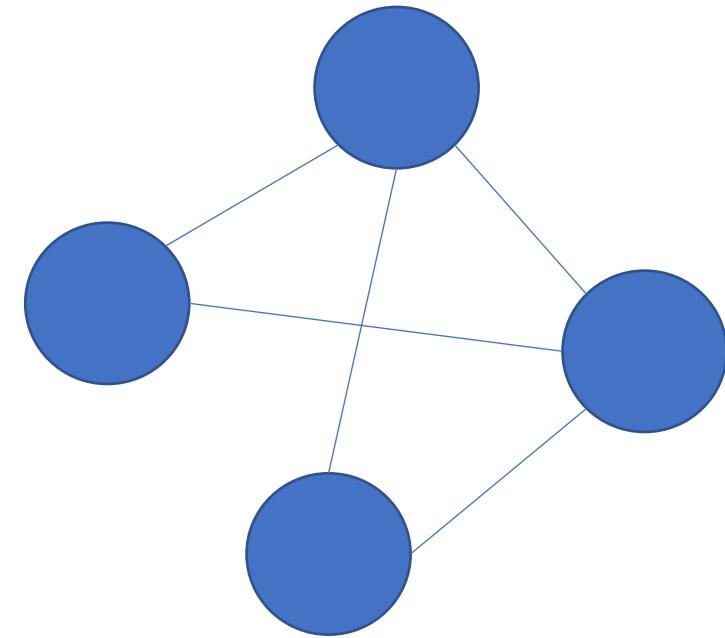
$$x_i(t+1) = \frac{x_i(t) + x_j(t)}{2}$$

Pentru push-pull, adițional:

$$x_j(t+1) = \frac{x_i(t) + x_j(t)}{2}$$

Ipoteze:

- Vecinul de intrare j al nodului i este selectat cu probabilitatea P_{ij}
- $P_{ij} > 0$ dacă $(i, j) \in E$ și $\sum_{j \in N(i)} P_{ij} = 1$



Algoritm gossip de medie

Fiecare nod $i \in V$ stochează valoarea $x_i(0)$. Notăm media $\mu = \frac{1}{n} \sum_i x_i(0)$.

Algoritm gossip *push-pull*: alege aleator (i, j) o pereche din E

$$x_i(t+1) = \frac{x_i(t) + x_j(t)}{2}, \quad x_j(t+1) = \frac{x_i(t) + x_j(t)}{2}.$$

Sau echivalent,

$$x(t+1) = W(t)x(t)$$

unde (e_i coloana i a matricii identitate)

$$W(t) = W_{ij} = I - \frac{1}{2}(e_i - e_j)(e_i - e_j)^T.$$

Matricea ponderilor este dinamică și aleatoare.

Algoritm gossip de medie

$$x(t+1) = W(t)x(t), \quad W(t) = W_{ij} = I - \frac{1}{2}(e_i - e_j)(e_i - e_j)^T$$

Exemplu: $n = 4, i = 2, j = 3$

$$W_{23} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 \\ 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matricile W_{ij} sunt dublu stohastice, i.e. $W_{ij}\mathbf{1} = \mathbf{1}$, $\mathbf{1}^T W_{ij} = \mathbf{1}^T$. Mai mult: $W(t)^T W(t) = W(t)$.

Observăm că: $x(t)$ este un proces stochastic în care

- $x(1)$ depinde de alegerea (i_0, j_0)
 - $x(2)$ depinde de alegerea $H(2) = \{(i_0, j_0), (i_1, j_1)\}$
 - ...
 - $x(t)$ depinde de alegerea $H(t) = \{(i_0, j_0), (i_1, j_1), \dots, (i_{t-1}, j_{t-1})\}$
- ⇒ Urmărim convergența lui $x(t)$ în medie, i.e. $E[\sigma^2(x(t))|x(0)] \leq \epsilon$.

Algoritm gossip de medie

Notăm $y(t) = x(t) - \mu\mathbf{1}$, atunci varianță: $\sigma^2(x(t)) = \|y(t)\|^2$ (conservarea masei).

Observăm că

$$y(t) \perp \mathbf{1} \quad (\text{i.e. } y(t)^T \mathbf{1} = 0)$$

și atunci

$$y(t+1) = W(t)y(t).$$

Algoritm gossip de medie

Pentru a mărgini media varianței avem:

$$\begin{aligned} E[y(t+1)^T y(t+1) | x(0)] &= E[y(t)^T W(t)^T W(t) y(t) | x(0)] \\ &= E[y(t)^T W(t) y(t) | x(0)] \\ &= E[E_{ij}[y(t)^T W(t) y(t)] \mid x(t-1), \dots, x(0)] \\ &= E[y(t)^T E_{ij}[W(t)] y(t) \mid x(t-1), \dots, x(0)] \end{aligned}$$

unde $W = E_{ij}[W(t)] = I - \frac{1}{2n}D + \frac{P+P^T}{2n}$, $D_i = \sum_j P_{ij} + P_{ji}$.

$$E[y(t+1)^T y(t+1) | x(0)] = E[y(t)^T W y(t) | x(0)]$$

Matricea W este dublu stochastică. Folosind Perron-Frobenius, observăm că $\mathbf{1}$ este vector propriu maximal al W și deci $y(t)$ este ortogonal pe subspațiul generat de acest v. p. ($y(t)$ se află în subspațiul generat de restul de $n-1$ v.p. ai matricii W). Această proprietate indică

$$y(t)^T W y(t) \leq \lambda_2(W) y(t)^T y(t) = \lambda_2(W) \sigma^2(x(t)).$$

Algoritm gossip de medie

Varianța convergen liniar cu factorul $\lambda_2(W)$:

$$\begin{aligned} E[\sigma^2(x(t+1))|x(0)] &\leq \lambda_2(W)E[\sigma^2(x(t))|x(0)] \\ &\leq \lambda_2(W)^t \sigma^2(x(0)) \end{aligned}$$

Pentru o acuratețe fixată ϵ și P dat, după $O\left(\frac{3\log(\epsilon^{-1})}{\log(\lambda_2(W)^{-1})}\right)$ iterații este garantat: $E[\sigma^2(x(t))] \leq \epsilon$.

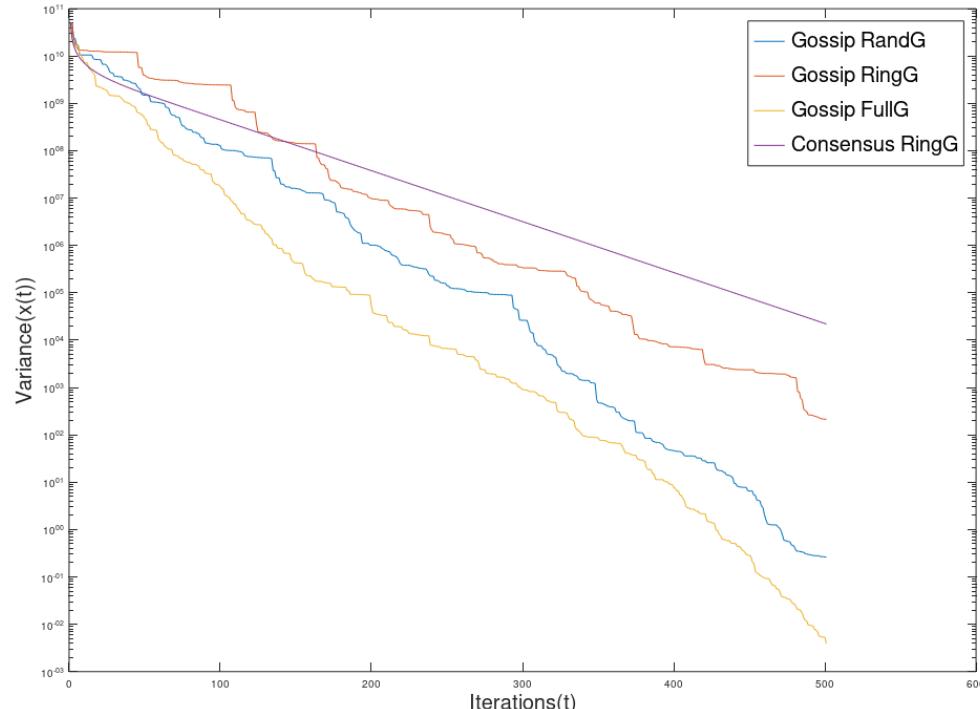
Exemplu:

$$n = 20$$

Inel: $\lambda_2(W) = 0.9988$

Random: $\lambda_2(W) = 0.9636$

Full: $\lambda_2(W) = 0.9500$



Algoritm gossip de medie

Comparatie cu Algoritmul Flooding (vezi Consens)

Exemplu:

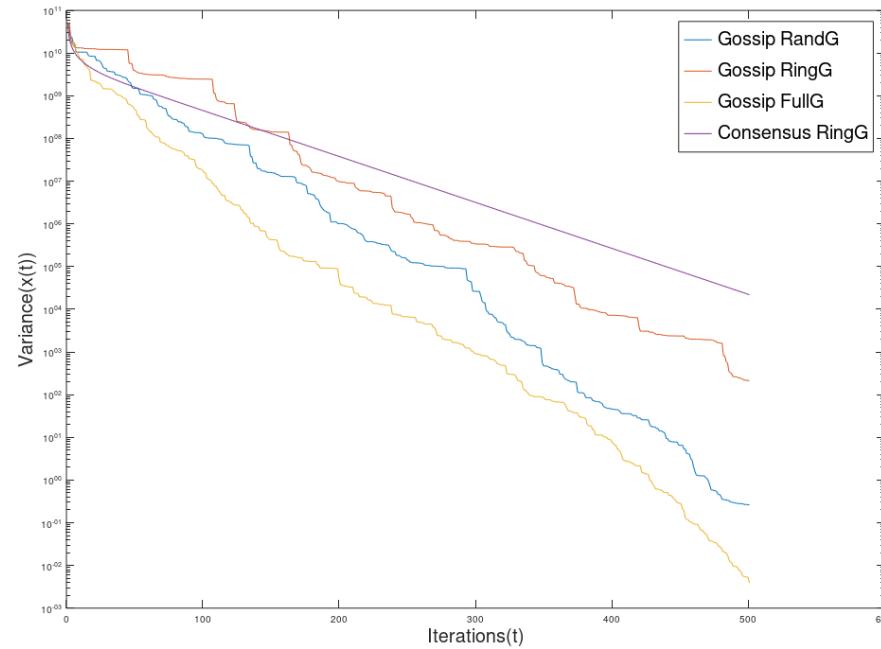
$$n = 20$$

Inel: $\lambda_2(W) = 0.9988$

Random: $\lambda_2(W) = 0.9636$ ($p = 0.8$)

Full: $\lambda_2(W) = 0.9500$

Vs. Alg. Flooding (inel)



Algoritm gossip de medie

Comparatie cu Algoritmul Flooding

Exemplu:

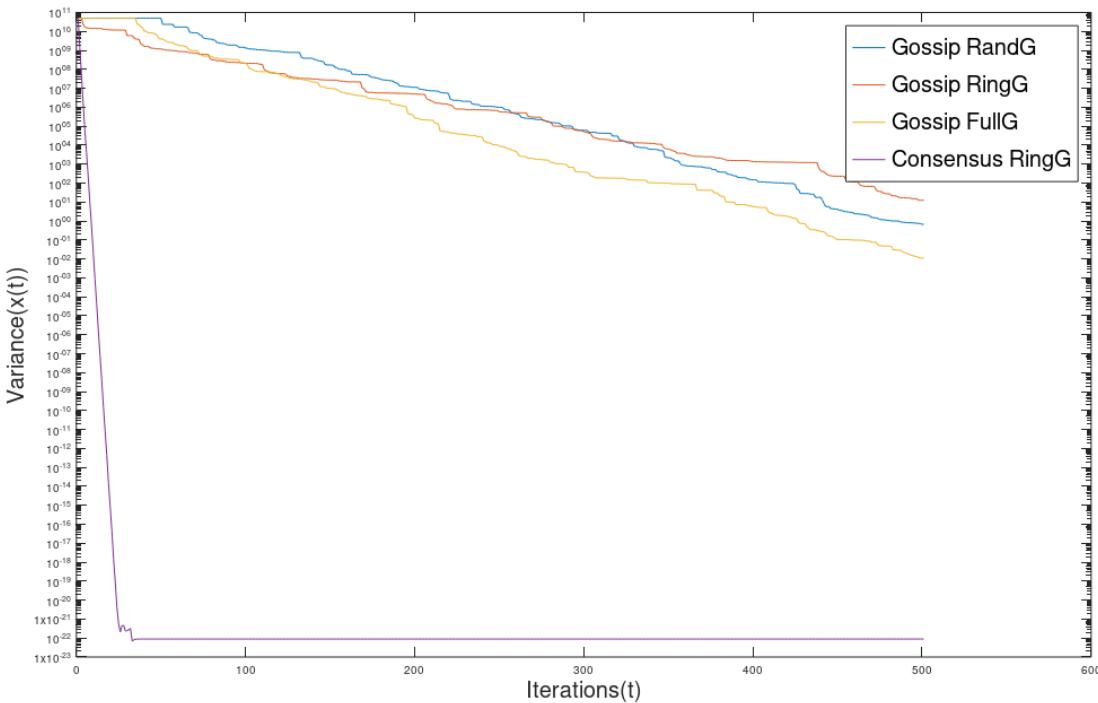
$$n = 20$$

Inel: $\lambda_2(W) = 0.9988$

Random: $\lambda_2(W) = 0.9636$ ($p = 0.8$)

Full: $\lambda_2(W) = 0.9500$

Vs. Alg. Flooding (graf random, $p = 0.8$)



Sisteme liniare

Sisteme liniare

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad \quad \quad \dots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

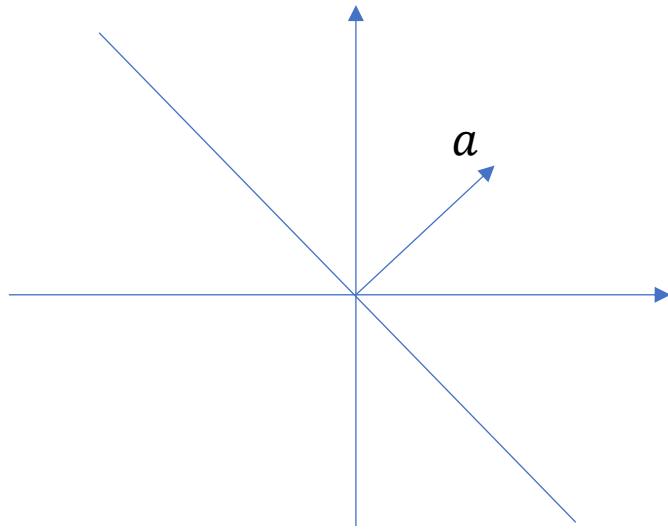
$$Ax = b$$

Hiperplan

În R^n , un hiperplan reprezintă un subspațiu liniar de dimensiune $n - 1$ parametrizat de $a \in R^n, b \in R$
 $H = \{x \in R^n : a^T x = b\}$.

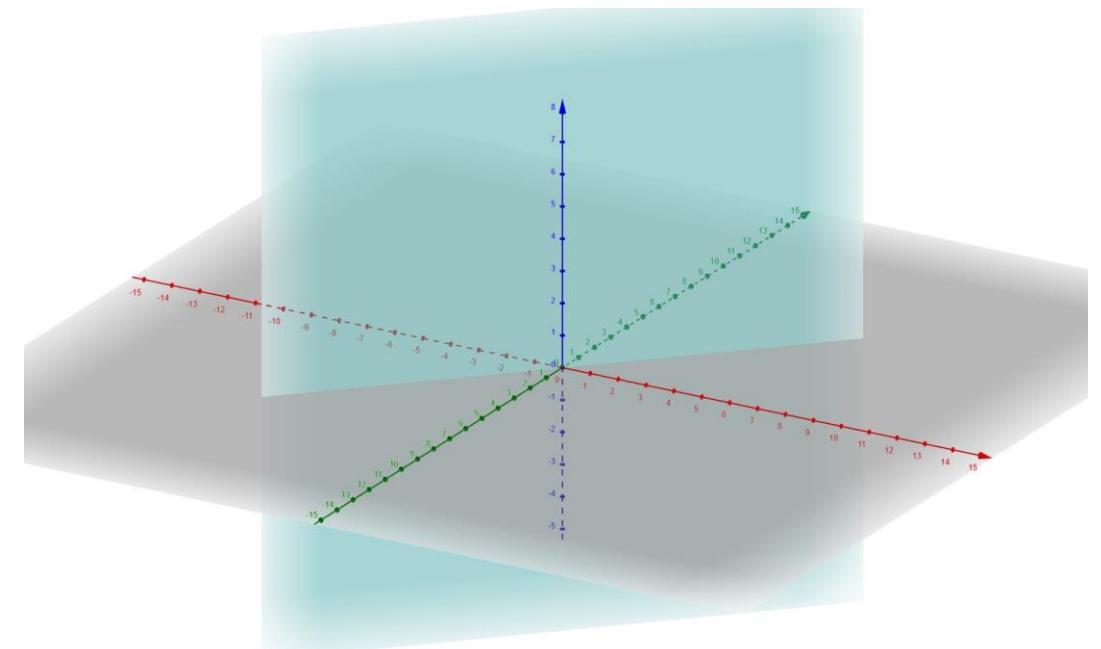
Exemplu:

- în $R^2 : H = \{x \in R^2 : x_1 + x_2 = 0\}$



Exemplu R^3 :

$$H = \{x \in R^3 : x_1 - x_2 = 0\}$$



Sistem liniar subdeterminat

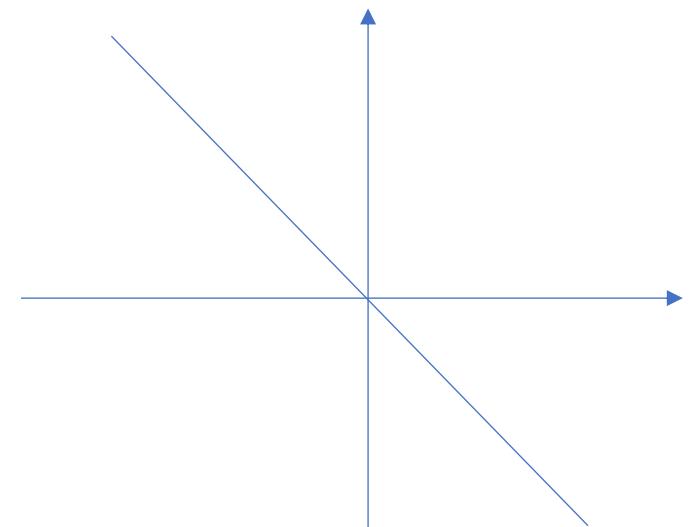
Un sistem liniar se numește subdeterminat

$$Ax = b, \quad A \in R^{m \times n}, b \in R^m$$

dacă $m < n$.

De exemplu, determinarea unui punct din H ,
presupune rezolvare unui sistem subdeterminat.
În acest caz, întregul H este spațiul soluțiilor.

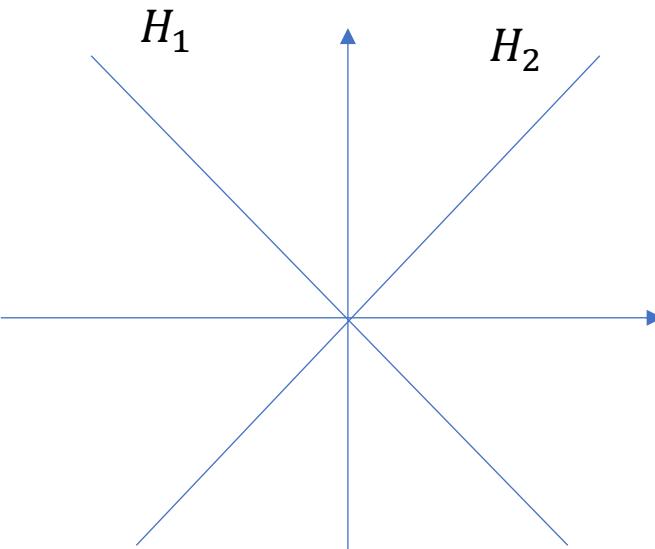
$$H = \{x \in R^2 : x_1 + x_2 = 0\}$$



Sistem liniar determinat

Un sistem liniar se numește determinat $Ax = b$, $A \in R^{m \times n}$, $b \in R^m$, dacă $m = n$.

De exemplu, avem sistemul cu două ecuații: $\begin{cases} x_1 + x_2 = 0 \\ x_1 - x_2 = 0 \end{cases}$



Fiecare dintre ecuații se exprimă printr-un hiperplan:

$$H_1 = \{x \in R^2 : x_1 + x_2 = 0\}$$
$$H_2 = \{x \in R^2 : x_1 - x_2 = 0\}$$

Observăm că

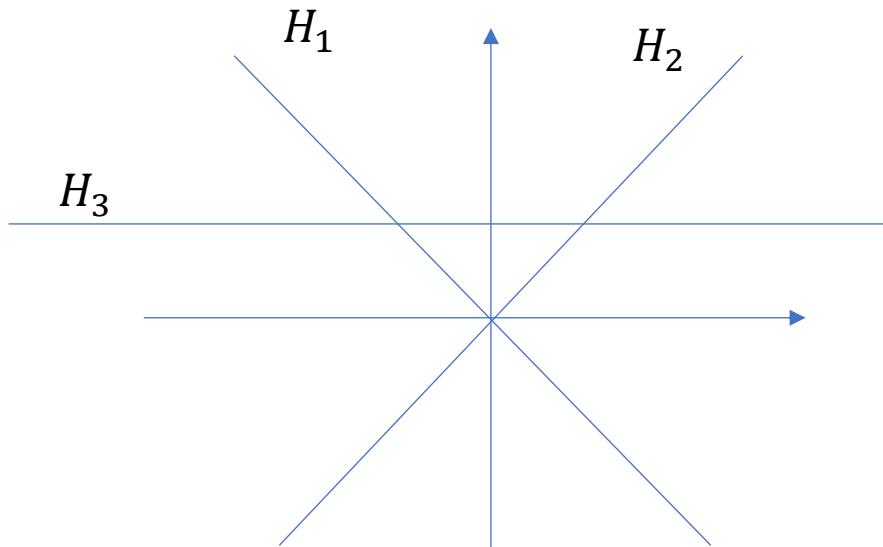
$$\{0\} = H_1 \cap H_2.$$

Sistem liniar supradeterminat

Un sistem liniar se numește supradeterminat $Ax = b, A \in R^{m \times n}, b \in R^m$ dacă $m > n$.

De exemplu, adăugând o ecuație sistemului precedent, obținem unul supradeterminat:

$$\begin{cases} x_1 + x_2 = 0 \\ x_1 - x_2 = 0 \\ x_2 = 1 \end{cases}$$



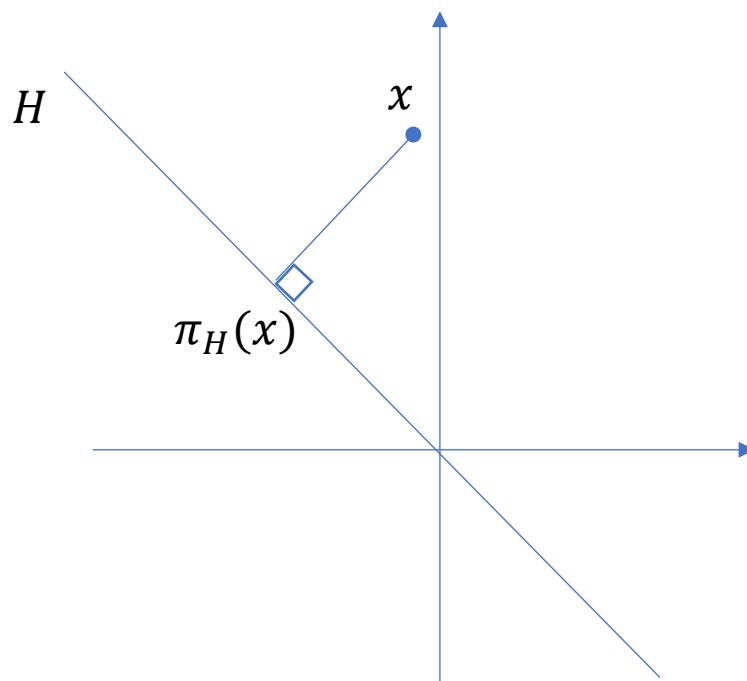
Sistemul este echivalent cu intersecția $H_1 \cap H_2 \cap H_3$, unde

$$\begin{aligned} H_1 &= \{x \in R^2 : x_1 + x_2 = 0\} \\ H_2 &= \{x \in R^2 : x_1 - x_2 = 0\} \\ H_3 &= \{x \in R^2 : x_2 = 1\}. \end{aligned}$$

Sistemul nu are soluție
 $\{\emptyset\} = H_1 \cap H_2 \cap H_3$.

Proiecție ortogonală

În R^n , proiecția ortogonală a punctului x pe hiperplanul $H = \{x \in R^n : a^T x = b\}$ este punctul $\pi_H(x)$ din H cel mai „apropiat” (în norma euclidiană) de x .

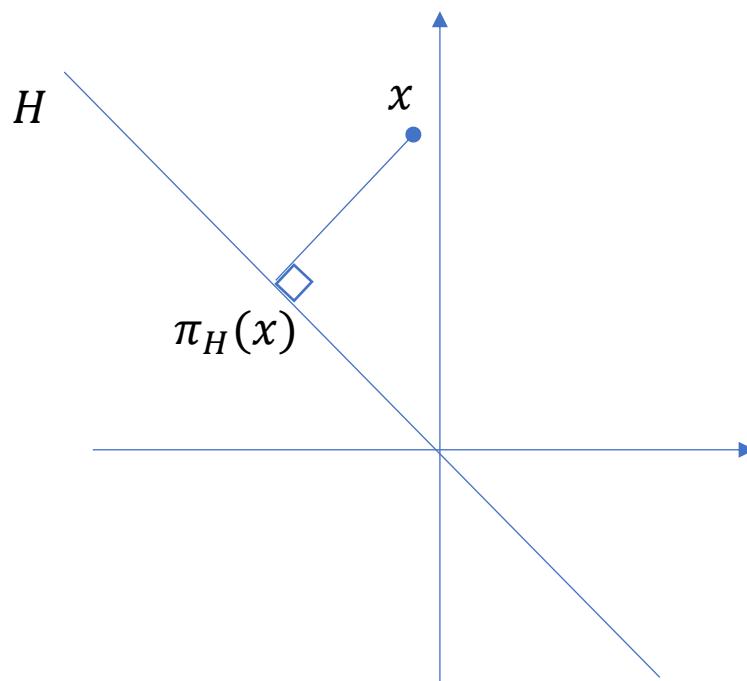


Proprietăți:

- $\|\pi_H(x) - x\| \leq \|z - x\|, \forall z \in H$
- $\pi_H(x) = x, \forall x \in H$
- $\pi_H(x)$ unică (H mulțime convexă)
- Formă explicită: $\pi_H(x) = x - \frac{a^T x - b}{\|a\|^2} a$

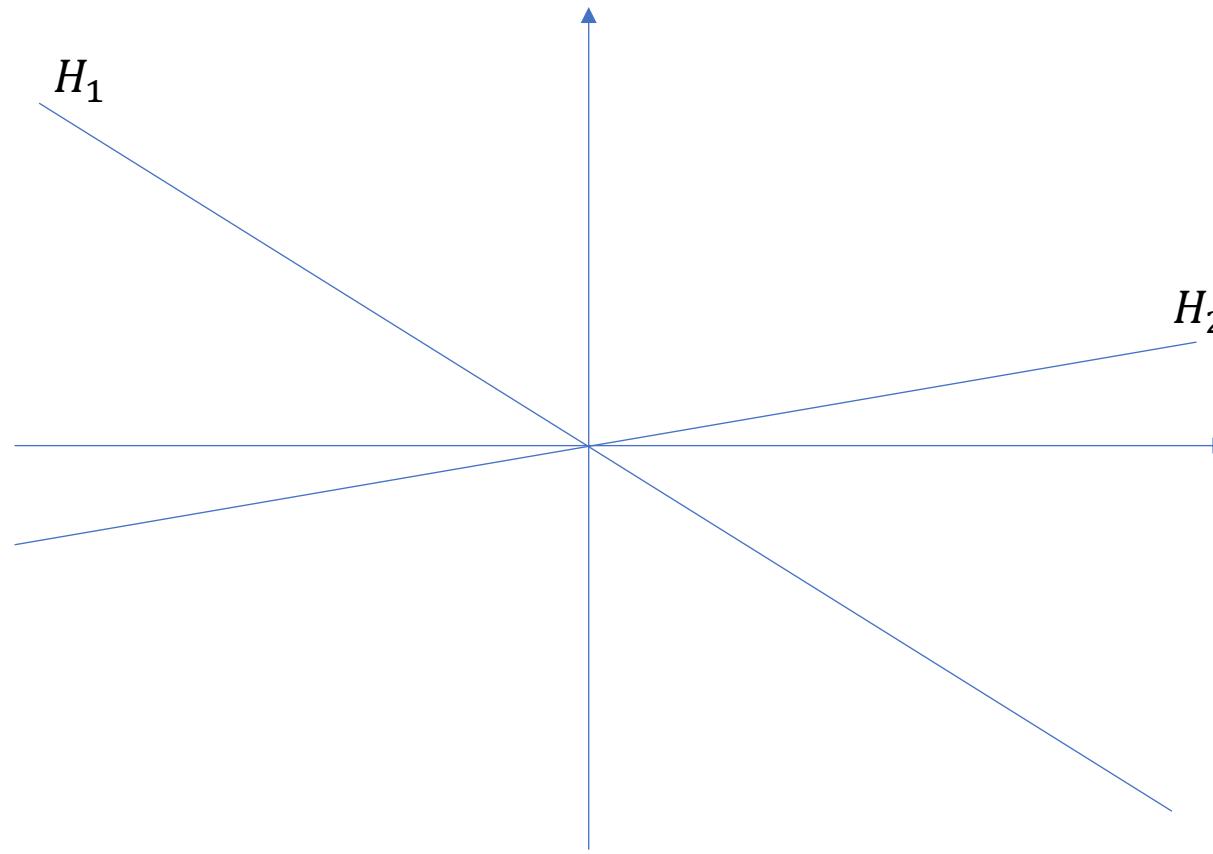
Proiecție ortogonală

Fie $H = \{x \in R^2 : x_1 + x_2 = 0\}$ și $z = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, u = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$. Calculați $\pi_H(z), \pi_H(u)$ și distanțele până la H .



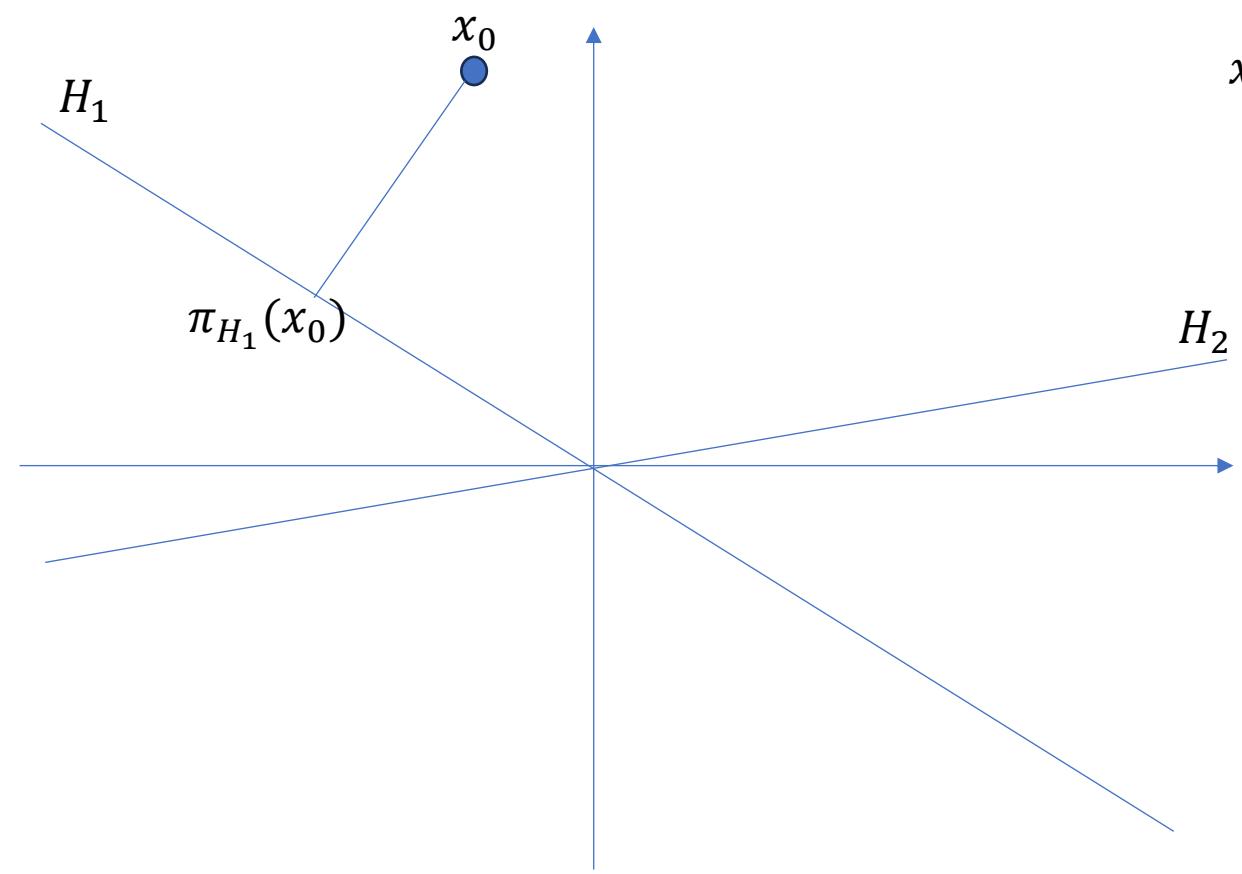
Algoritmul Proiecțiilor Alternative

Problemă: Fie $a_1, a_2 \in R^n$, $a_1 \neq a_2$. Rezolvați sistemul $\begin{cases} a_1^T x = b_1 \\ a_2^T x = b_2 \end{cases}$ folosind doar proiecții ortogonale.



Algoritmul Proiecțiilor Alternative

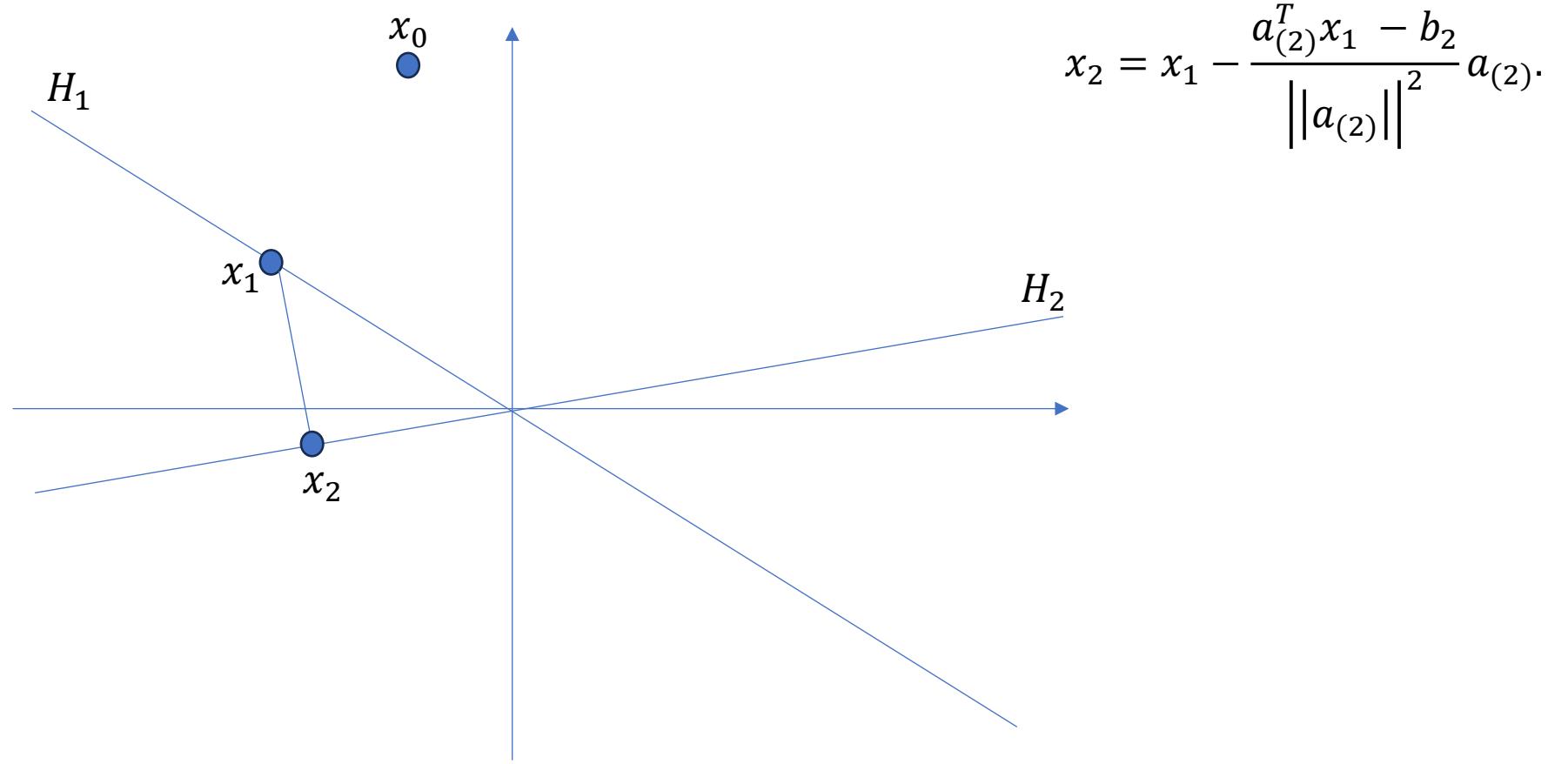
Alegem un x_0 arbitrar, alegem un hiperplan (H_1 sau H_2) și realizăm sirul iterativ $x_{t+1} = \pi_{H_{i_t}}(x_t)$.



$$x_1 = x_0 - \frac{a_{(1)}^T x_0 - b_1}{\|a_{(1)}\|^2} a_{(1)}.$$

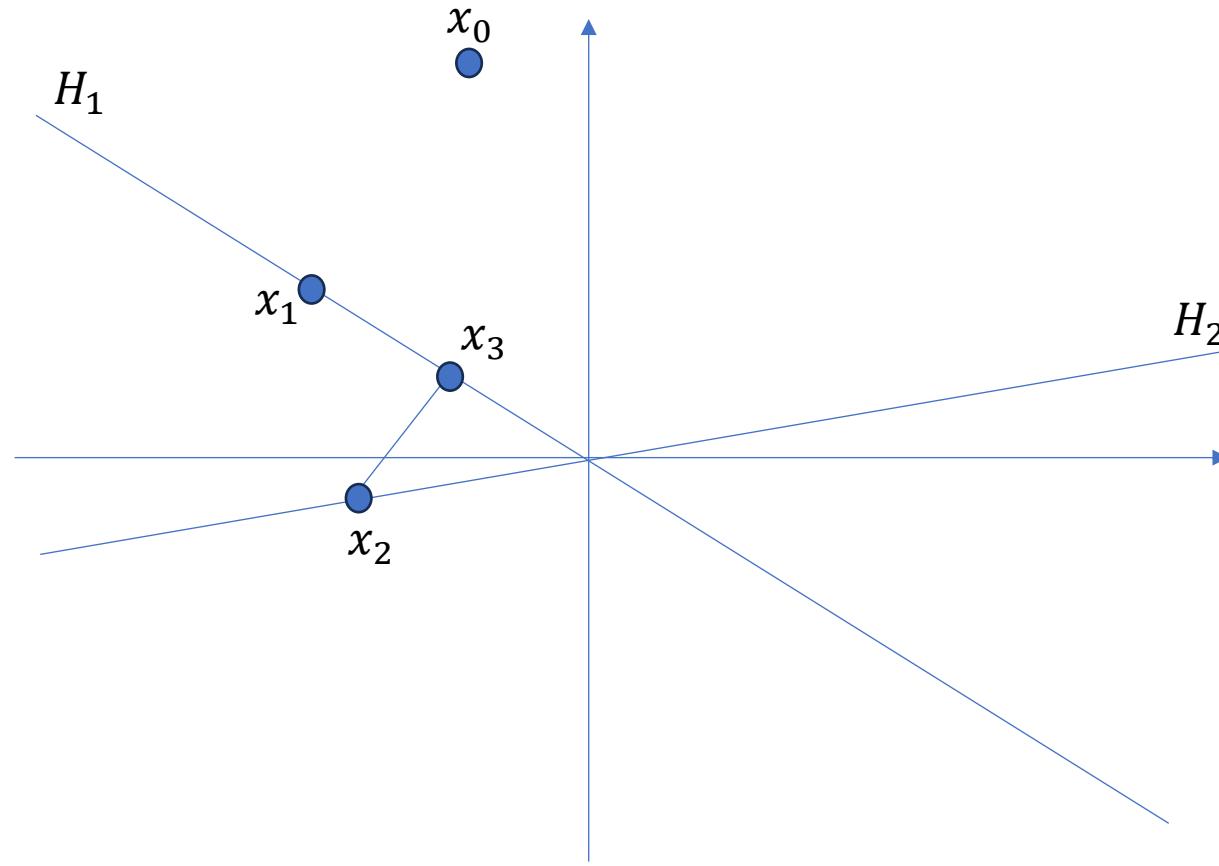
Algoritmul Proiecțiilor Alternative

Alegem un x_0 arbitrar, alegem un hiperplan (H_1 sau H_2) și realizăm sirul iterativ $x_{t+1} = \pi_{H_{i_t}}(x_t)$.



Algoritmul Proiecțiilor Alternative

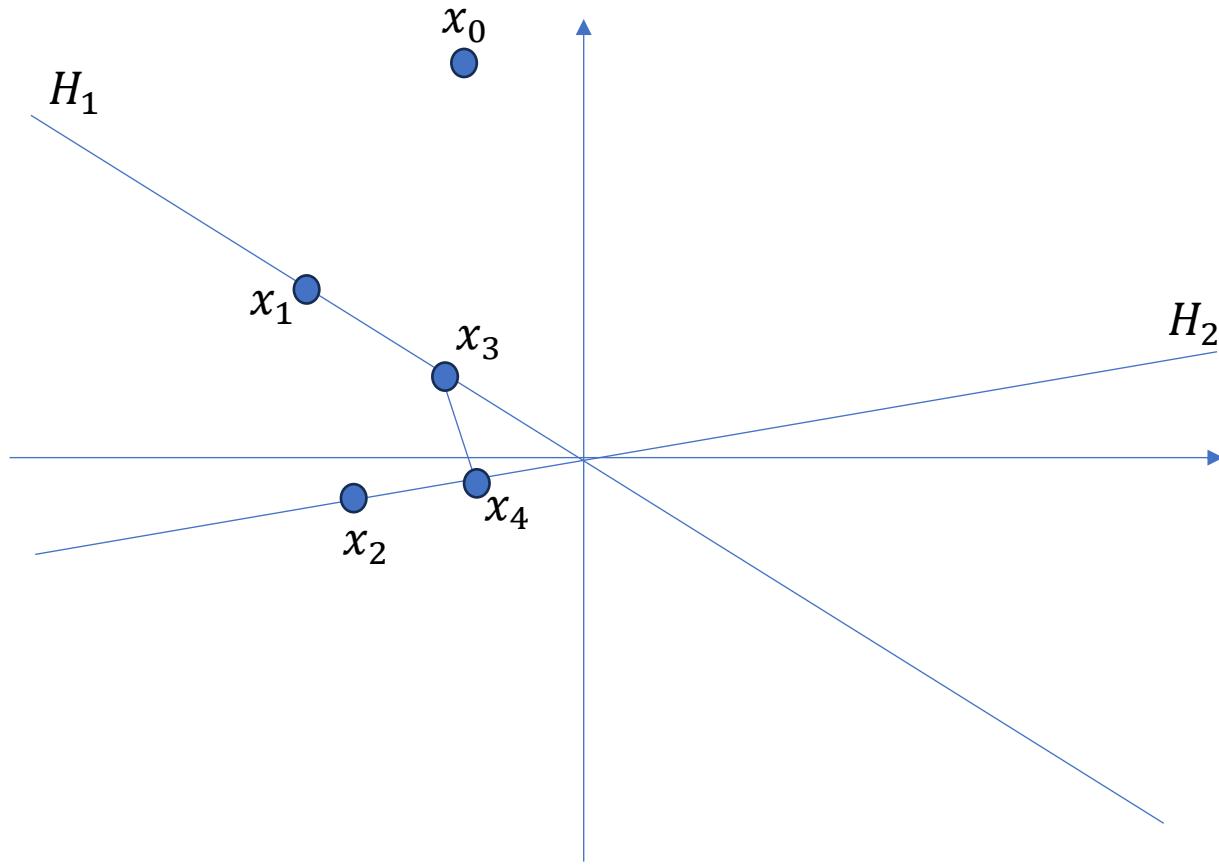
Alegem un x_0 arbitrar, alegem un hiperplan (H_1 sau H_2) și realizăm sirul iterativ $x_{t+1} = \pi_{H_i_t}(x_t)$.



$$x_3 = x_2 - \frac{a_{(1)}^T x_2 - b_1}{\|a_{(1)}\|^2} a_{(1)}.$$

Algoritmul Proiecțiilor Alternative

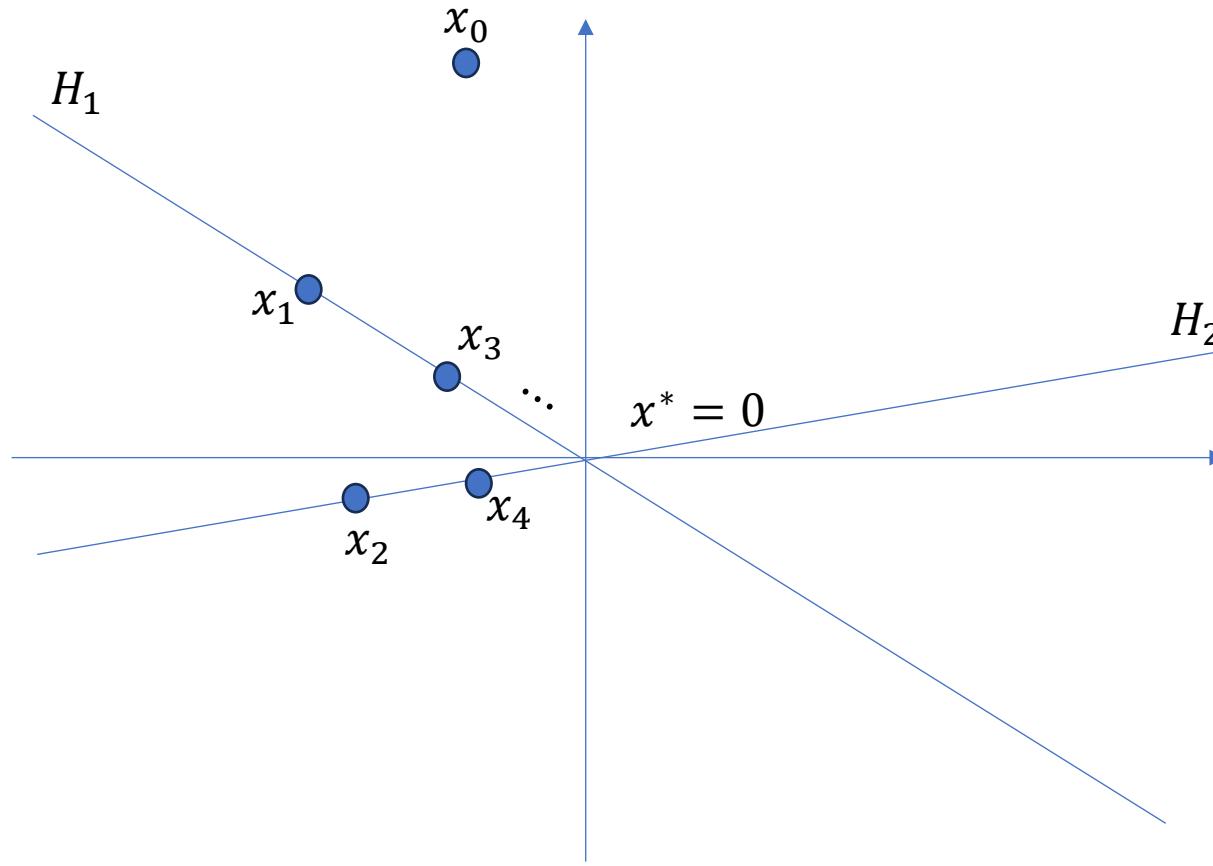
Alegem un x_0 arbitrar, alegem un hiperplan (H_1 sau H_2) și realizăm sirul iterativ $x_{t+1} = \pi_{H_{i_t}}(x_t)$.



$$x_4 = x_3 - \frac{a_{(2)}^T x_3 - b_2}{\|a_{(2)}\|^2} a_{(2)}.$$

Algoritmul Proiecțiilor Alternative

Alegem un x_0 arbitrar, alegem un hiperplan (H_1 sau H_2) și realizăm sirul iterativ $x_{t+1} = \pi_{H_{i_t}}(x_t)$.



Algoritmul Proiecțiilor Alternative

Convergența pt 2 hiperplane:

$$x_{t+1} = \pi_{H_1}(x_t) = \pi_{H_1}(\pi_{H_2}(x_{t-1}))$$

Observăm: $x_{t+1}, x_{t-1} \in H_1$ și $x_t, x_{t-2} \in H_2$, de aceea $a_{(1)}^T x_{t-1} = b_1$.

Reziduul $\|Ax_t - b\|_2$ măsoară distanța la $H_1 \cap H_2$, deci

$$a_{(2)}^T x_{t+1} - b_2 = (a_{(2)}^T x_{t-1} - b_2) \frac{(a_{(1)}^T a_{(2)})^2}{\|a_{(1)}\|^2 \|a_{(2)}\|^2} - (a_{(1)}^T x_{t-1} - b_1) \frac{a_{(1)}^T a_{(2)}}{\|a_{(1)}\|^2}$$

$$a_{(2)}^T x_{t+1} - b_2 = (a_{(2)}^T x_{t-1} - b_2) \frac{(a_{(1)}^T a_{(2)})^2}{\|a_{(1)}\|^2 \|a_{(2)}\|^2}$$

$$\|Ax_{t+1} - b\|_2 = \frac{(a_{(1)}^T a_{(2)})^2}{\|a_{(1)}\|^2 \|a_{(2)}\|^2} \|Ax_{t-1} - b\|_2$$

Factorul de convergență $\frac{|a_{(1)}^T a_{(2)}|}{\|a_{(1)}\| \|a_{(2)}\|}$ = cosinusul unghiului dintre H_1 și H_2 .

Algoritmul Proiecțiilor Alternative

Rezolvarea sistemului

$$Ax = b, \quad A \in R^{m \times n}, b \in R^m$$

este echivalentă cu determinarea

$$x \in H_1 \cap H_2 \cap \dots \cap H_m,$$

unde $H_i = \{x: a_{(i)}^T x = b_i\}$. Schema generală APA selectează la iterată t un hiperplan $H_{i(t)}$ și actualizează iterată:

$$x_{t+1} := \pi_{H_{i(t)}}(x_t)$$

Alegerea $H_{i(t)}$ poate urma regula *ciclică, aleatoare, „greedy”* etc.

Algoritmul Proiecțiilor Alternative

Formatul distribuit al problemei găsirii $x \in H_1 \cap H_2 \cap \dots \cap H_m$ presupune determinarea $z \in H_i = \{x: a_{(i)}^T x = b_i\}$ o sarcină individuală asociată nodului P_i .

Deci presupunem că P_i poate rezolva propria sarcină locală (i.e. determinarea unui punct din H_i , respectiv proiecția ortogonală pe H_i).

Starea $x_i(t) \in R^n$ reprezintă estimarea locală a soluției la pasul t . Dacă starea nodului P_i se află pe hiperplanul H_i , i.e. $x_i \in H_i$, atunci fiecare P_i va satisface ecuația $a_{(i)}^T x_i = b_i$, dar nu avem o soluție a întregului sistem.

Algoritmul de Consens Proiectat (ACP)

Algoritmul de Consens Proiectat compune pasul de consens cu cel de proiecție ortogonală:

$$x_i(t+1) = \pi_{H_i} \left(\sum_{j \in \mathcal{N}_i^- \cup i} w_{ij} x_j(t) \right) \quad \forall i,$$

unde ponderile $w_{ij} \geq 0$, $\sum_{j \in \mathcal{N}_i^- \cup i} w_{ij} = 1$ (medie).

Dacă $H_i = \mathbb{R}^n$, atunci ACP se reduce la Algoritmul Flooding de medie (din cursul 6):

$$x_i(t+1) = w_{ii} x_i(t) + \sum_{j \in \mathcal{N}_i^-} w_{ij} x_j(t) \quad \forall i.$$

Algoritmul de Consens Proiectat (ACP)

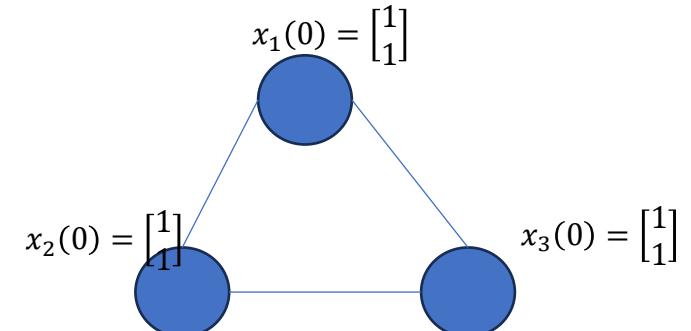
Algoritmul de Consens Proiectat compune pasul de consens cu cel de proiecție ortogonală:

$$x_i(t + 1) = \pi_{H_i} \left(\sum_{j \in \mathcal{N}_i^- \cup i} w_{ij} x_j(t) \right) \quad \forall i.$$

- La fiecare iterare P_i își menține starea pe H_i (păstrează activă ecuația $a_{(i)}^T x = b_i$) prin operația de proiecție.
- P_i se apropie de consens prin operația de medie $w_i^T x(t)$
- Este suficient consensul pentru a garanta rezolvarea sistemului.

Algoritmul de Consens Proiectat (ACP)

- Considerăm $n = 3, w_{ij} = \frac{1}{3}$ (uniforme)
- Rezolvăm sistemul: $\begin{cases} x_1 + x_2 = 0 \\ x_1 - x_2 = 0 \\ 2x_1 + 3x_2 = 0 \end{cases}$, pornind din $x_i(0) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, i = 1,2,3.$



Algoritmul de Consens Proiectat (ACP)

$$x_1(1) = \pi_{H_1} \left(\frac{1}{3}x_1(0) + \frac{1}{3}x_2(0) + \frac{1}{3}x_3(0) \right)$$

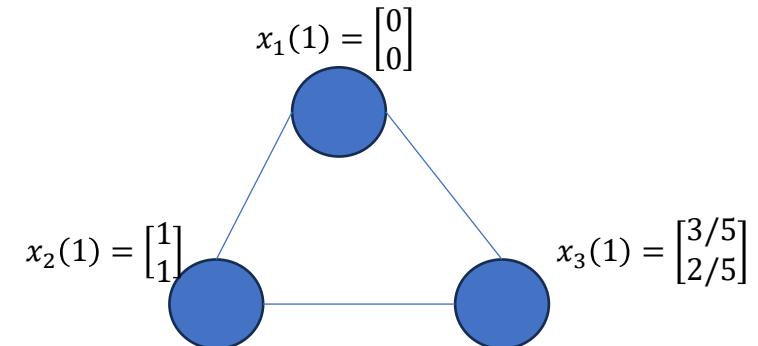
$$\begin{aligned} x_1(1) &= \pi_{H_1} \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\ &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned}$$

$$x_2(1) = \pi_{H_2} \left(\frac{1}{3}x_1(0) + \frac{1}{3}x_2(0) + \frac{1}{3}x_3(0) \right)$$

$$\begin{aligned} x_2(1) &= \pi_{H_2} \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\ &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{aligned}$$

$$x_3(1) = \pi_{H_3} \left(\frac{1}{3}x_1(0) + \frac{1}{3}x_2(0) + \frac{1}{3}x_3(0) \right)$$

$$\begin{aligned} x_3(1) &= \pi_{H_3} \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\ &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \frac{5}{25} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3/5 \\ 2/5 \end{bmatrix} \end{aligned}$$



Algoritmul de Consens Proiectat (ACP)

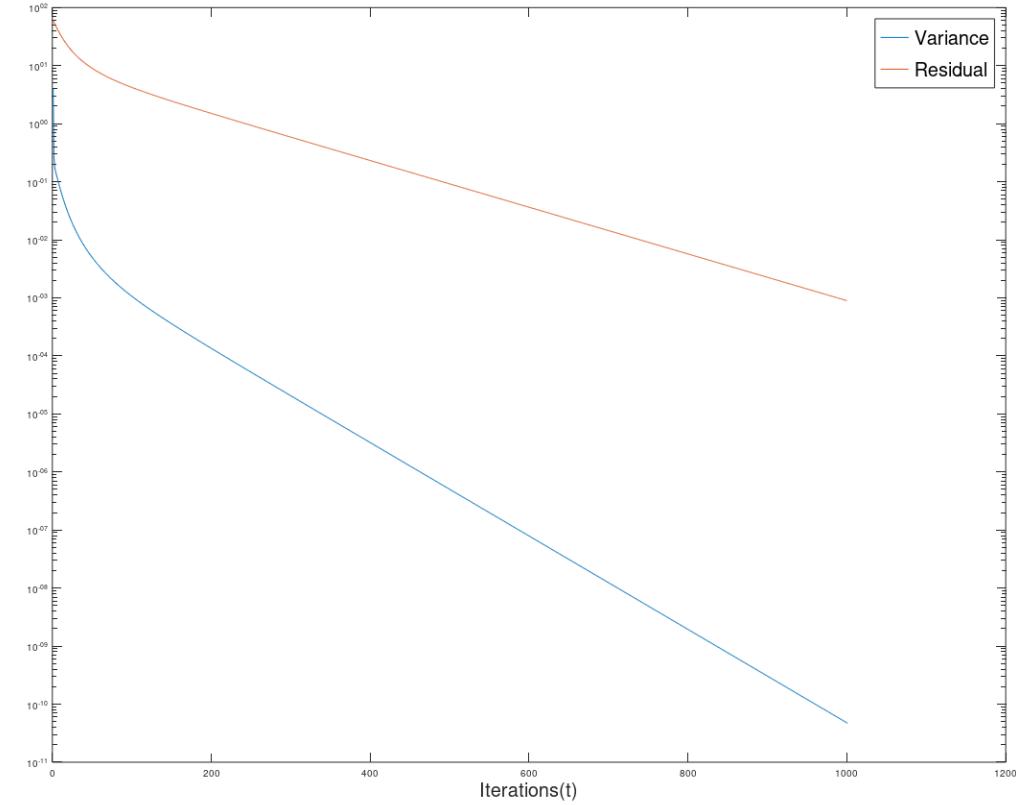
Algoritmul de Consens Proiectat compune pasul de consens cu cel de proiecție ortogonală:

$$x_i(t + 1) = \pi_{H_i} \left(\sum_{j \in \mathcal{N}_i^- \cup i} w_{ij} x_j(t) \right) \quad \forall i$$

Teorema. Fie matricea ponderilor W dublu stochastică. Presupunem că există constantă $\eta > 0$ astfel încât toate ponderile $w_{ij} > 0$ satisfac $w_{ij} \geq \eta$ ($w_{ii} \geq \eta$). Dacă sistemul $Ax = b$ are soluție, atunci sirul generat de ACP atinge consensul asimptotic (într-una dintre soluțiile sistemului).

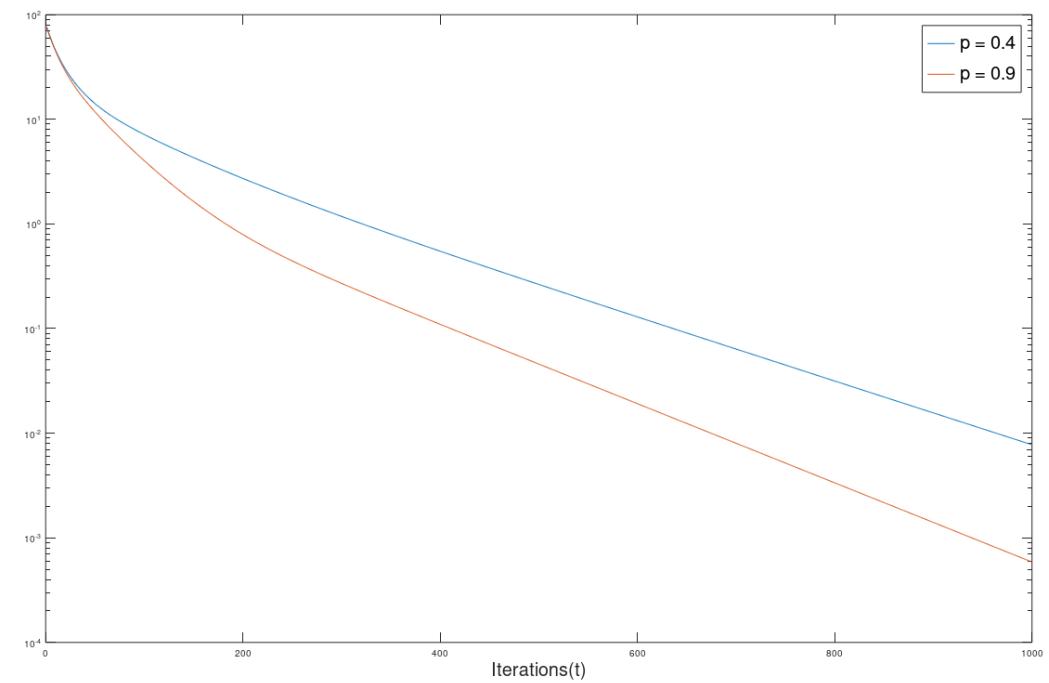
Algoritmul de Consens Proiectat (ACP)

- Date aleatoare (functia Matlab randn)
 $A \in R^{20 \times 50}, W \in R^{20 \times 20}$
- Graf random, probabilitatea de apariție a unei muchii
 $p = 0.9$
- Curbele urmăresc:
 $\sum_i var(x_i(t))$ (blue), $\sum_i ||Ax_i(t) - b||$ (red)



Algoritmul de Consens Proiectat (ACP)

- Date aleatoare (functia Matlab randn)
 $A \in R^{20 \times 50}, W \in R^{20 \times 20}$
- Grafuri aleatoare, probabilitatea de apariție a unei muchii $p = 0.4$, respectiv $p = 0.9$
- Curbele urmăresc: $\sum_i ||Ax_i(t) - b||$



References

- [1] Nedic, Angelia, Asuman Ozdaglar, and Pablo A. Parrilo. "Constrained consensus and optimization in multi-agent networks." *IEEE Transactions on Automatic Control* 55.4 (2010): 922-938.
- [2] S. Boyd, A. Gosh, B. Prabhakar, D. Shah, *Gossip Algorithms: Design, Analysis and Applications*, *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*. Vol. 3. IEEE, 2005.
- [3] Márk JELASITY. *Gossip-based protocols for large-scale distributed systems*. PhD Thesis, 2013.