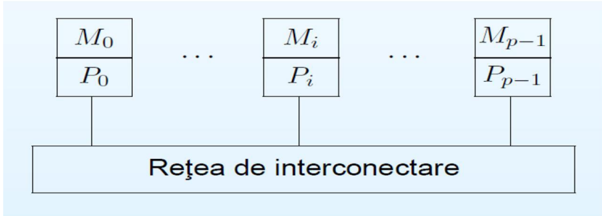
Sistem distribuit

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.

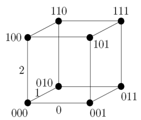
Sistem MIMD cu memorie distribuită

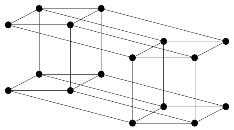
- Fiecare procesor are memorie proprie (arhitectura locală cu RISC şi memorie ierarhică, de obicei)
- Comunicaţia se face printr-o reţea de comunicaţie, prin mesaje explicite
- Operaţii favorizate: paralele, la nivel de bloc
- Comunicaţia prin mesaje necesită algoritmi dedicaţi



Sistem paralel (Cluster computing)

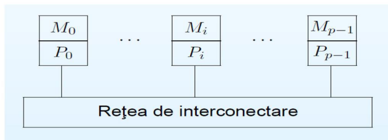
- **Informații globale disponibile**: număr de noduri ale rețelei, topologia rețelei, distribuția datelor în rețea, indexarea globală a nodurilor
  - Control asupra distribuției datelor
  - Control asupra execuției locale per nod
  - Control asupra implicării nodurilor în rețea
- **Timp de comunicație** inter-noduri neglijabil/mărginit (apropiere geografică)
  - Sincronizare: ceas fizic comun
- Topologie statică
- Probabilitatea scăzută a defectelor
- Complexitatea timp vs. complexitatea mesaj





Comunicația prin mesaje: Modelul SPMD

- MIMD cu memorie distribuiă
- 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;
- Deşi toate procesoarele văd acelaşi program, instrucţiunile executate nu sunt identice



Modelul SPMD

- Procesoarele sunt numerotate $0, \ldots, p$
- 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

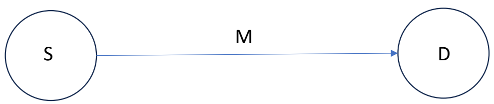
1. rank ← adresa proprie
2. **dacă** rank = 0 **atunci**
   1. a ← 1
3. **altfel**
   1. a ← 0

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 inițializează a cu 0 pe toate componentele, cu excepția primei componente (care este 1).

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

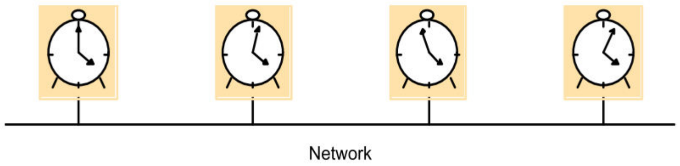
Timp

- 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

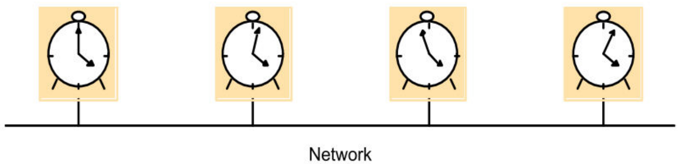
- In SD, ceasurile locale sunt **decalate** și **întârziate**
- **Decalaj** între nodurile (i, j) la momentul t: $|C_i(t) - C_j(t)|$
- **Întârziere** între nodurile (i, j) la momentul t: $|\frac{d}{dt}C_i(t) - \frac{d}{dt}C_j(t)|$



Problema sincronizării

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

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



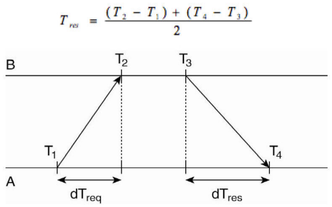
Algoritmul lui Cristian

- 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 referința
- Nodurile care nu sunt referința se vor sincroniza cu ceasul referinței

Algoritmul lui Cristian

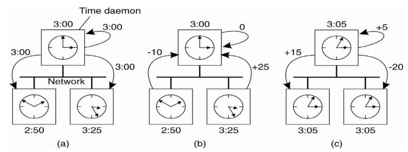
Procedura AC: Nodul P cere periodic valoarea timpului de la R:

- 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 $T_{res} \approx \frac{T_{req}+T_{res}}{2}$



Algoritmul de medie (Berkeley Algorithm)

- Referința este unul din nodurile rețelei, ales eventual prin proceduri de leader-election
- Restul nodurilor urmăresc alinierea ceasurilor cu referința (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 distribuie ajustările pentru $C_i(t)$



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

$$\max\{id_1, id_2, \dots, id_n\}$$

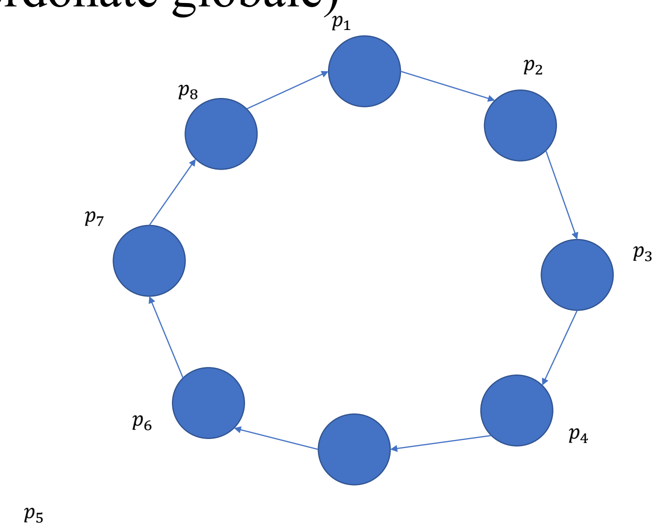
Alegere Lider (coordonate globale)

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)

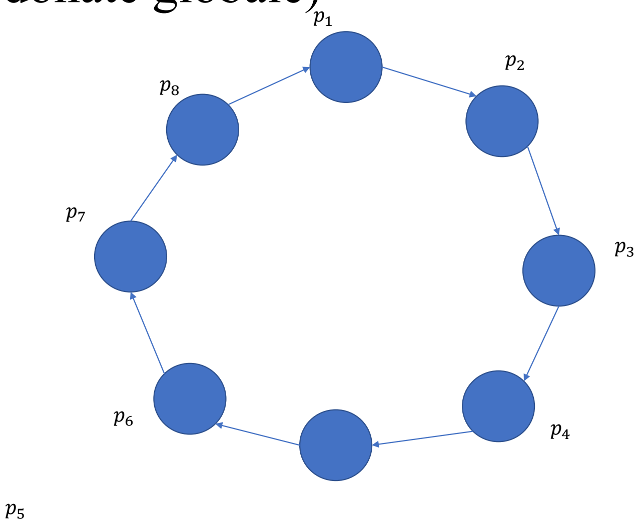
M_i: - int n (număr noduri)
- int i (index propriu)
- int id (id propriu)
- int id_max (id propriu)

% 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)

Memorie locală nod i:
- int n (număr noduri)
- int i (index propriu)
- int id (id propriu)
- int id_max (id propriu)

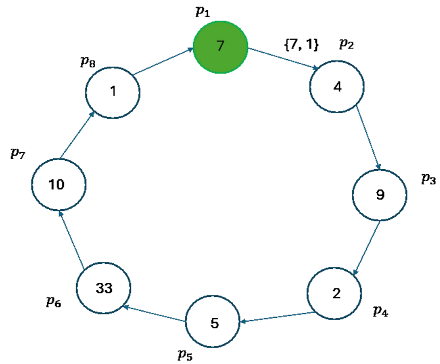
% Faza I: Max ID
…
% Faza II: Difuzare Max ID (Broadcast)

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);



Algoritmul Flooding (LCR)

Algoritm **Flooding**():

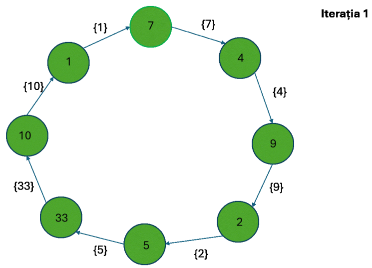
M_i: - int id (id propriu)
- int send_id (var auxiliară), inițial id
- status ∈ {lider, non-lider}, inițial non-lider

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 iterații până la anunțarea unui lider, iar complexitatea mesaj este $O(n^2)$**.**



Algoritmul Flooding (graf tare conectat)

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

M_i: - int *id* (id propriu)
- int *max_id* (var auxiliară), inițial id
- *status* ∈ {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} ∪ U)
4. **If** (rounds == diam):
   1. **If** (max_id == id): status = leader;
   2. **Else**: status = non-leader;
5. **Else**: send(max_id, 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*;

Gerard Le Lann. *Distributed systems – toward a formal approach*. In Bruce Gilchrist, editor, Information Processing 77, Proceedings of IFIP Congress, 155-160, 1977.
Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

Gerard Le Lann. *Distributed systems – toward a formal approach*. In Bruce Gilchrist, editor, Information Processing 77, Proceedings of IFIP Congress, 155-160, 1977.
Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

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 $< v_j, id_j >$ 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.

Kuhn, Fabian, Nancy Lynch, and Rotem Oshman, *Distributed computation in dynamic networks*. Proceedings of the forty-second ACM symposium on Theory of computing, 2010.
Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

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 $< v_j, id_j >$ 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 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()).

Kuhn, Fabian, Nancy Lynch, and Rotem Oshman, *Distributed computation in dynamic networks*. Proceedings of the forty-second ACM symposium on Theory of computing, 2010.
Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

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

Hendrickx, Julien M., and John N. Tsitsiklis. "Fundamental limitations for anonymous distributed systems with broadcast communications." *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2015.

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

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

Hendrickx, Julien M., and John N. Tsitsiklis. "Fundamental limitations for anonymous distributed systems with broadcast communications." *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2015.

Teorema de imposibilitate pentru rețele anonime

Teorema de imposibilitate [Hendricx&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.

Hendrickx, Julien M., and John N. Tsitsiklis. "Fundamental limitations for anonymous distributed systems with broadcast communications." *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2015.

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* ∈ {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*} ∪ U)
4. **If** (t == diam):
   1. **If** (*max_v* == *v*): status = leader;
   2. **Else**: status = non-leader;
5. **Else**: 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.

Gerard Le Lann. *Distributed systems – toward a formal approach*. In Bruce Gilchrist, editor, Information Processing 77, Proceedings of IFIP Congress, 155-160, 1977.
Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

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

Exemple:

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

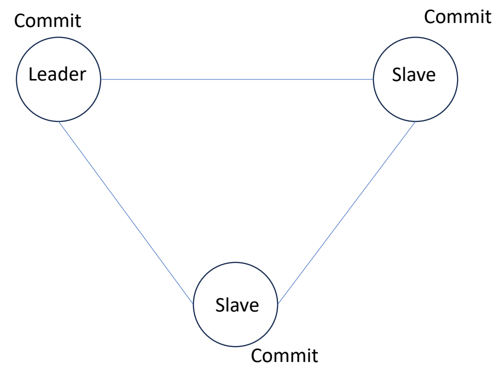
Consens centralizat

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

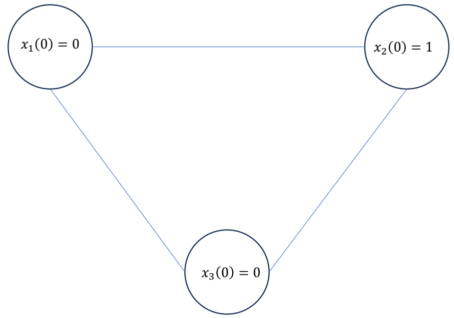


Consens distribuit

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 $e.g. n, diam(G), G$
- 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. buf_i = Gather($\mathcal{N}_i$);
2. $x_i = \begin{cases} 1, dacă\ |\{buf_i[j] = 1\}| \geq \left\lceil \frac{|\mathcal{N}_i|}{2} \right\rceil \\ 0, dacă\ |\{buf_i[j] = 1\}| < \left\lceil \frac{|\mathcal{N}_i|}{2} \right\rceil \end{cases}$
3. Broadcast(x_i, $\mathcal{N}_i$);



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.

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 $< v_j, id_j >$ 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.

Algoritm Flooding pentru consens majoritar

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

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

Funcție transformare nod i ():

1. Fie 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** $\frac{1}{2}\left(1 + \mathrm{sgn}\left(x(t) - \frac{1}{2}\right)\right)$
4. **Else**: send($x(t+1)$, $\mathcal{N}_i^+$)
5. $t := t+1$

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

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 mulțimea 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** $\frac{1}{2}\left(1 + \mathrm{sgn}\left(x(t) - \frac{1}{2}\right)\right)$
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:

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

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:

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

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

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

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 stohastică pe linii, atunci se atinge consensul asimptotic, i.e. $x(t) \to c\mathbf{1}$ când $t \to \infty$. În plus, dacă matricea A este stohastică 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 stohastică pe linii (fiecare să realizeze la fiecare iterație o combinație convexă între starea proprie și stările vecinilor)
- Dacă matricea ponderilor este, în plus, stohastică pe coloane, atunci valoarea de consens este media aritmetică a stărilor inițiale.

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.

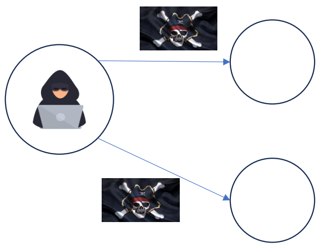
Defectul Crash

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

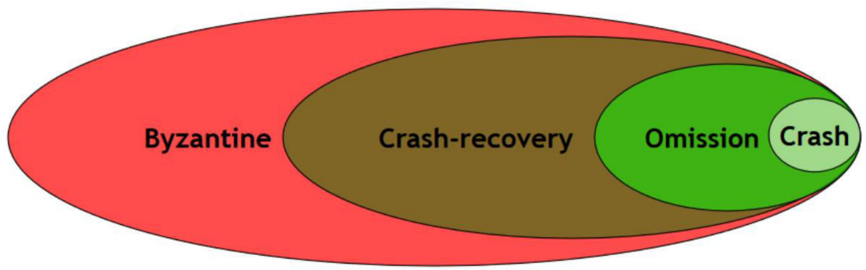
Defect Bizantin

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

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



Ierarhie



Seif Haridi, Distributed Systems course, https://canvas.instructure.com/courses/902299/files/31539484?module_item_id=6616396

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 *corect* din sistem va decide eventual la un moment de timp.

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

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 $< v_j, id_j >$ 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. **If** ($t > s+1$):
   1. **Return** $f(M_i$(t))
7. $t := t+1$

- Paradigma de ‘’robustificare’’ a algoritmilor distribuiti
- 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 $t \le t' \le f+1$ 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 dupa $s+1$ 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 (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 $< v_j, id_j >$ 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**: 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, dupa $s+1$ seturi de $diam(G)$ iterații, există cel puțin unul în care niciun nod nu are defect. Însa $diam(G)$ sunt suficiente pentru a atinge consensul între nodurile corecte.

Valori și vectori proprii

Definiție. Valorile proprii (*eigenvalues*) ale matricii $A \in R^{n\times n}$ sunt date de n rădăcini ale polinomului caracteristic $p(z) = \det(zI_n - A)$. Mulțimea 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 satisface:

$$Ax = \lambda x$$

și vector propriu *la stânga* dacă satisface:

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

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 (matrici stohastice)

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

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

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

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

În plus, dacă matricea A este *stohastică pe coloane*, i.e. $\mathbf{1}^\mathbf{T} A = A^T$, 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: $x(\infty) = \lim_{t\in\infty} A^t\, x(0) =\ c\mathbf{1}$ 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.

Algoritmul ByzFlood

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

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

Funcție transformare nod i ():

% Runda 1

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

% 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 are primit cel puțin n/2 voturi pentru aceeași valoare.

În cele 3 situații P_i și P_j ajung la consens.

Consens distribuit (cu procese defecte)

Starea (valoarea) uniformă a nodurilor unui sistem distribuit.

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

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

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

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

Ordonarea evenimentelor

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

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

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

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

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

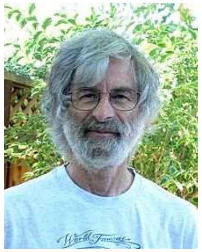
Ordine cauzală

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

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

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

Ceasuri logice Lamport



Mecanism introdus de Leslie Lamport în 1978.

- Ceas logic = marcaj de timp C asociat unui eveniment

- Fiecare P_i întreține un ceas local C_i (scalar, care reflectă percepția locală și globală). La fiecare eveniment local (de calcul) $C_i = C_i + d \;\; (d > 0)$.

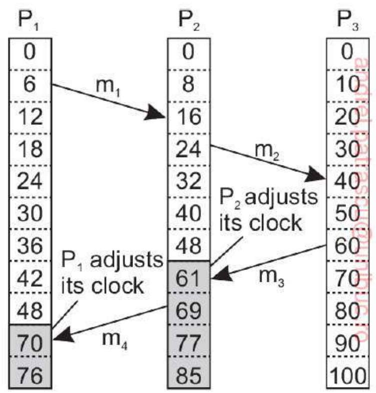
- De asemenea, la fiecare eveniment de comunicație $P_i \rightarrow_m P_j$
  - P_i atașează mesajului m valoarea curentă locală a ceasului C_i
  - P_j recepționează mesajul m și execută: $C_j \coloneqq \max\{C_j, C_{msg}\}, C_j \coloneqq C_j + 1$

Ceasuri logice Lamport

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

P_1 ajustează ceasul său local folosind timpul primit de la P_2

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

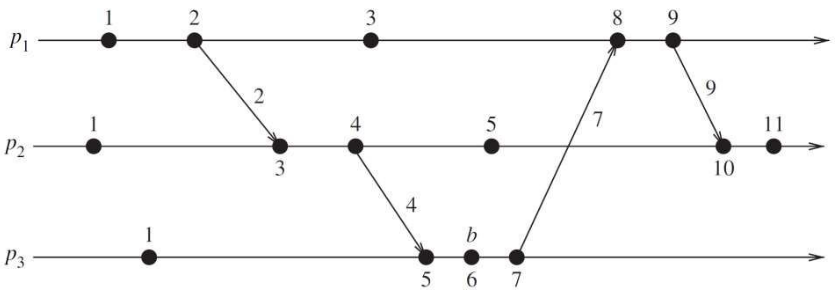


Ceasuri logice Lamport

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

Nu are loc consistența tare: $C(a) < C(b)$ nu implică a $\rightarrow b$
Actualizarea unui ceas scalar nu reține valorile de timp ale vecinilor!

Ceasuri logice Lamport



Ceasuri vectoriale

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

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

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

Noua actualizare:

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

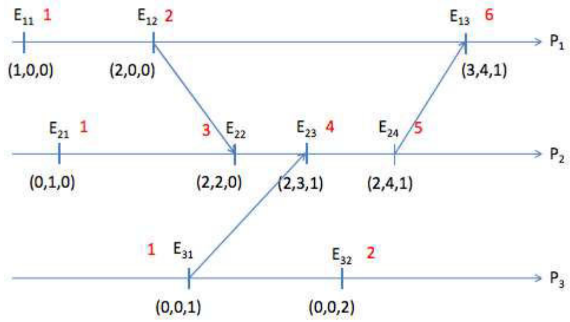
Când m este livrat de P_i la P_j atașează V_i la mesajul m

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

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

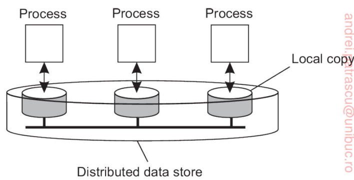
Ceasuri vectoriale

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



Sisteme distribuite cu memorie partajată

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

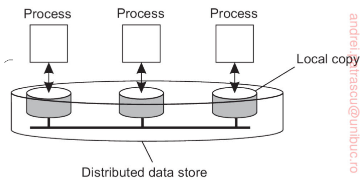


Ipoteze:
- O replică (copie) a memoriei partajate este menținută local de fiecare nod.
- O operație de citire-scriere poate avea loc din oricare nod al sistemului.
- Prin rețeaua de comunicație, operația se propagă în celelalte replici.

Consistența: replicile converg asimptotic către consensul (persistent).

Sisteme distribuite cu memorie partajată (SDMP)

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



Modele de consistență

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

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

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

andrei.trascu@unibuc.ro
andrei.trascu@unibuc.ro

Modele de consistență

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

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

Consistența strictă (linearizabilitate)

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

În particular, necesită ca toate nodurile sa dețină:

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

Consistența strictă (linearizabilitate)

a) Respectă consistența strictă
b) Nu respectă consistența strictă

Imposibil de implementat într-un SDMP real

Consistența secvențială

Model de consistență mai relaxat decât consistența strictă.

Cerință:

Toți clienții văd operațiile de scriere în aceeași ordine:

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

Consistența secvențială

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

Consistența secvențială

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

Pentru aceste operații avem CS

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

Consistența secvențială

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

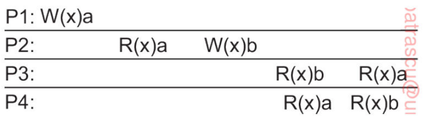
Ordering of operations	**Result**	
$W_1(x)a$; $W_1(y)a$; $W_2(y)b$; $W_2(x)b$	$R_1(x)b$	$R_2(y)b$
$W_1(x)a$; $W_2(y)b$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a$; $W_2(y)b$; $W_2(x)b$; $W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_1(x)a$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_1(x)a$;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

Consistența cauzală

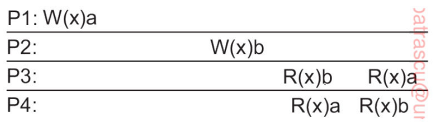
- Relaxează mai departe cerințele consistenței secvențiale.
- Două operații sunt în relație cauzală dacă:
  - o citire este urmată de o scriere în același client
  - o scriere a unui obiect este urmată de o citire a aceluiași obiect în orice client
- Operațiile de scriere care sunt potențial cauzale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

Consistența cauzală

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

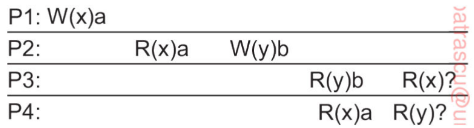


(a)



(b)

Consistența cauzală



Operația $R_3(x)$

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

Operația $\mathrm{R}_4(x)$

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

Consistență eventuală

Sub concurență slabă, cerințele de consistență sunt slabe.

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

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

Algoritmi

Două scheme simple pentru a păstra consistența secvențială:

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

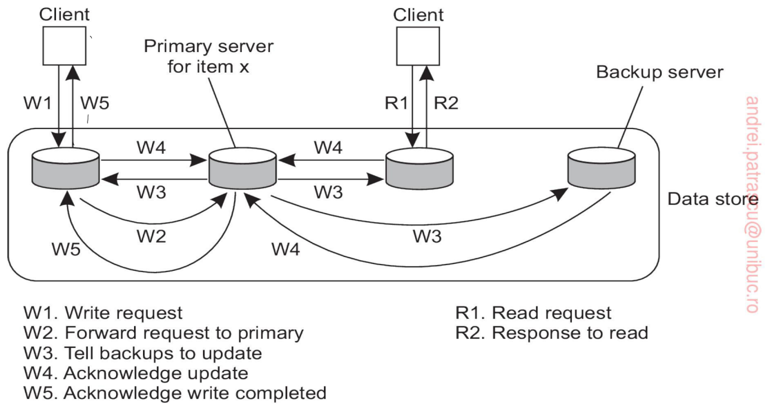
Schema Remote-write

Toate operațiile de scriere sunt executate pe un singur nod-server (distant).
Acest model este asociat cu arhitecturile tradiționale client-server.

Algoritm:

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

Schema Remote-write



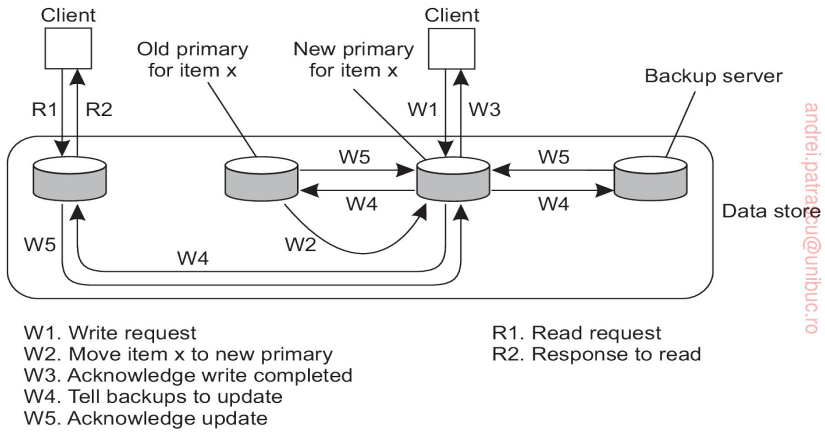
andrei.patrascu@unibuc.ro

Schema Local-write

O singură copie a elementului x este actualizată.

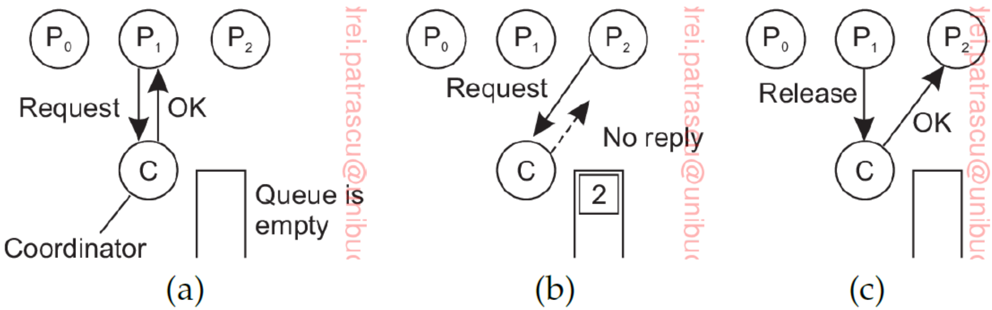
- La operația de scriere, elementul x va fi transferat la replica care realizează operațiile de scriere (primary)
  - Sunt posibile multiple scrieri succesive executate local
- Starea de "primară" a unei replici este transferabilă

Schema Local-write



andrei.patrascu@unibuc.ro

Excludere mutuală centralizată



Excludere mutuală centralizată

Avantaje:

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

Dezavantaje:

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

Excludere mutuală distribuită

Idee: Putem folosi ceasurile logice Lamport pentru ordonarea solicitărilor?

Premisă: Fiecare proces P_i păstrează un ceas logic L_i.

Algoritmul Ricart-Agrawala:

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

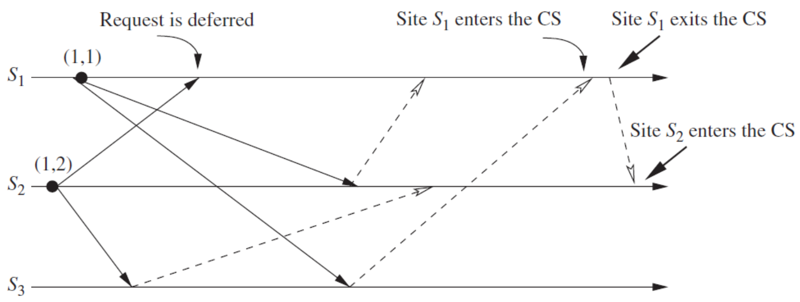
Excludere mutuală distribuită

Premisă: Fiecare proces P_i păstrează un ceas logic L_i.

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

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

Excludere mutuală distribuită



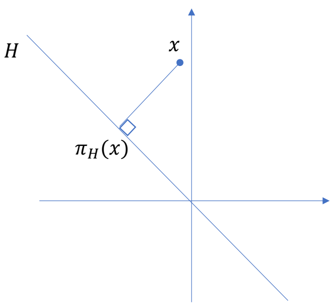
Excludere mutuală distribuită

Analiză:

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

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

Algoritmul de Consens Proiectat (ACP)

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

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

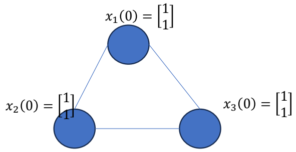
unde ponderile $\mathrm{w_{ij}} \ge 0, \sum_{j \in \mathcal{N}_i^- \cup i} w_{ij} = 1$ (medie).

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

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

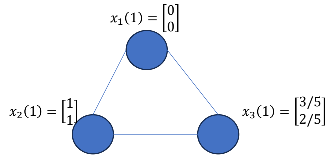
Algoritmul de Consens Proiectat (ACP)

- 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)

$$x_1(1) = \pi_{H_1}\left(\frac{1}{3}x_1(0) + \frac{1}{3}x_2(0) + \frac{1}{3}x_3(0)\right)$$
$$x_1(1) = \pi_{H_1}\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
$$x_2(1) = \pi_{H_2}\left(\frac{1}{3}x_1(0) + \frac{1}{3}x_2(0) + \frac{1}{3}x_3(0)\right)$$
$$x_2(1) = \pi_{H_2}\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$x_3(1) = \pi_{H_3}\left(\frac{1}{3}x_1(0) + \frac{1}{3}x_2(0) + \frac{1}{3}x_3(0)\right)$$
$$x_3(1) = \pi_{H_3}\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \frac{5}{25}\begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3/5 \\ 2/5 \end{bmatrix}$$



Algoritmul de Consens Proiectat (ACP)

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

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

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

Probleme

1. Fie sistemul $\begin{cases} x_1 + x_2 - x_3 = 1 \\ x_1 - x_2 + 2x_3 = 2 \\ 2x_1 + 3x_2 = 5 \end{cases}$. Rezolvă Algoritmul Proiecțiilor Alternative (serial) acest sistem liniar? Stabiliți un punct inițial x(0) și scrieți primele 3 iterații APA serial, cu regula de alegere ale hiperplanelor ciclică/dinamică.

Alegeri posibile:

- Ciclică: $i(0) = 1, i(1) = 2, \dots, i(m-1) = m$
- Aleatoare: $randint(m)$
- Dinamică („greedy”): $i(t) = argmax_i\ |a_i^T x(t) - b_i|$

Probleme

1. APA serial converge doar dacă există o soluție a sistemului $Ax = b$. Deci pentru a determina convergența asimptotică este necesar calculul unei soluții a sistemului
$$\begin{cases} x_1 + x_2 - x_3 = 1 \\ x_1 - x_2 + 2x_3 = 2 \\ 2x_1 + 3x_2 = 5 \end{cases}.$$

Alegere $x(0) = [0;0;0]^T$. Varianta ciclică:

$$x(1) = \pi_{H_1}\left(\begin{bmatrix}0\\0\\0\end{bmatrix}\right) = \begin{bmatrix}0\\0\\0\end{bmatrix} + \frac{1}{3}\begin{bmatrix}1\\1\\-1\end{bmatrix} = \begin{bmatrix}1/3\\1/3\\-1/3\end{bmatrix}$$

$$x(2) = \pi_{H_2}\left(\begin{bmatrix}1/3\\1/3\\-1/3\end{bmatrix}\right) = \begin{bmatrix}1/3\\1/3\\-1/3\end{bmatrix} + \frac{4}{9}\begin{bmatrix}1\\-1\\2\end{bmatrix} = \begin{bmatrix}7/9\\-1/9\\5/9\end{bmatrix}$$

$$x(3) = \pi_{H_3}\left(\begin{bmatrix}7/9\\-1/9\\5/9\end{bmatrix}\right) = \frac{1}{9}\begin{bmatrix}7\\-1\\5\end{bmatrix} + \frac{34}{117}\begin{bmatrix}2\\3\\0\end{bmatrix} = \begin{bmatrix}159/117\\21/117\\5/9\end{bmatrix}$$

Costul unei iterații?

Probleme

1.(continuare) Alegere $x(0) = [0;0;0]^T$. Varianta dinamică:

$$|a_1^T x(0) - b_1| = 1, |a_2^T x(0) - b_2| = 2, |a_3^T x(0) - b_3| = 5$$

$$x(1) = \pi_{H_3}\left(\begin{bmatrix}0\\0\\0\end{bmatrix}\right) = \begin{bmatrix}0\\0\\0\end{bmatrix} + \frac{1}{5}\begin{bmatrix}2\\3\\0\end{bmatrix} = \begin{bmatrix}2/5\\3/5\\0\end{bmatrix}$$

$$|a_1^T x(1) - b_1| = 0, |a_2^T x(1) - b_2| = \frac{11}{5}, |a_3^T x(1) - b_3| = 0$$

$$x(2) = \pi_{H_2}\left(\begin{bmatrix}2/5\\3/5\\0\end{bmatrix}\right) = \begin{bmatrix}2/5\\3/5\\0\end{bmatrix} + \frac{11}{30}\begin{bmatrix}1\\-1\\2\end{bmatrix} = \begin{bmatrix}23/30\\7/30\\22/30\end{bmatrix}$$

$$|a_1^T x(2) - b_1| = \frac{11}{15}, |a_2^T x(2) - b_2| = 0, |a_3^T x(2) - b_3| = \frac{83}{30}$$

$$x(3) = \pi_{H_3}\left(\begin{bmatrix}23/30\\7/30\\22/30\end{bmatrix}\right) = \frac{1}{30}\begin{bmatrix}23\\7\\22\end{bmatrix} + \frac{83}{390}\begin{bmatrix}2\\3\\0\end{bmatrix} = \begin{bmatrix}465/390\\340/390\\22/30\end{bmatrix}$$

Costul unei iterații?

Sistem distribuit asincron

Teorema imposibilitate. Într-un sistem distribuit asincron este imposibil de atins consensul (distribuit) chiar și sub un singur defect de tip *crash*.

Idea demonstrației: în cazul unui potențial defect de tip crash, nu este posibil să se distingă între un proces defect și unul corect cu întârzieri în comunicație.

Consecință. Toate problemele care se pot reduce la una de consens, sunt imposibil de rezolvat sub un singur defect de tip *crash, e.g.* alegere lider, calcul distribuit de funcții, difuzare sigură etc.

Algoritmi asincroni - formalizare

$$x_i(t+1) = f_i\left(x_1\left(\tau_1^i(t)\right), x_2\left(\tau_2^i(t)\right), \dots, x_n\left(\tau_n^i(t)\right)\right),$$

În general presupunem că fiecare P_i stochează o *vedere proprie* stării globale $x^i(t) = (x_1^i(t), x_2^i(t), \dots, x_n^i(t))$, pe baza căreia actualizează $x_i^i(t)$ la $t \in T^i$ prin relația:

$$x_i^i(t+1) = f_i(x^i(t))$$

Exemplu

Determinați x astfel încât

$$x_1 = \frac{1}{2}x_1 + \frac{1}{2}x_2$$
$$x_2 = \frac{1}{2}x_1 + \frac{1}{2}x_2$$

- Mulțimea soluțiilor satisface: $x_1 = x_2$.
- Sincron, iterația $x(t+1) = Ax(t)$, ajunge după 1 pas la optim:

$$x(1) = \begin{bmatrix} \frac{1}{2}(x_1(0) + x_2(0)) \\ \frac{1}{2}(x_1(0) + x_2(0)) \end{bmatrix}$$

Algoritmi asincroni - exemplu

Scenariu:

- Avem 2 procesoare P_1, P_2
- Comunică la anumite momente $\{\tau_1, \tau_2, \dots\}$
- Transmiterea/folosirea informatiei comunicate se face instantaneu.

$$P_1\colon\ x_1(t+1) = \frac{x_1(t)}{2} + \frac{x_2(\tau_k)}{2}, \qquad \tau_k \leq t < \tau_{k+1}$$
$$P_2\colon\ x_2(t+1) = \frac{x_1(\tau_k)}{2} + \frac{x_2(t)}{2}, \qquad \tau_k \leq t < \tau_{k+1}$$

Algoritmi asincroni - exemplu

$$P_1\colon\ x_1(t+1) = \frac{x_1(t)}{2} + \frac{x_2(\tau_k)}{2}, \qquad \tau_k \leq t < \tau_{k+1}$$

Între momentele τ_k și τ_{k+1}, P_1 menține $x_2(\tau_k)$ constant și execută iterația de mai sus de $\tau_{k+1} - \tau_k$ ori:

$$P_1\colon\ x_1(\tau_k+1) = \frac{1}{2}x_1(\tau_k) + \frac{1}{2}x_2(\tau_k)$$

Algoritmi asincroni - exemplu

$$P_1\colon x_1(t+1) = \frac{x_1(t)}{2} + \frac{x_2(\tau_k)}{2}, \qquad \tau_k \leq t < \tau_{k+1}$$

Între momentele τ_kși τ_{k+1}, P_1 menține $x_2(\tau_k)$ constant și execută iterația de mai sus de $\tau_{k+1} - \tau_k$ ori:

$$\begin{aligned} x_1(\tau_k+2) &= \frac{1}{2}x_1(\tau_k+1) + \frac{1}{2}x_2(\tau_k) \\ &= \frac{1}{2}\left[\frac{1}{2}x_1(\tau_k) + \frac{1}{2}x_2(\tau_k)\right] + \frac{1}{2}x_2(\tau_k) \\ &= \boldsymbol{\frac{1}{4}x_1(\tau_k) + \left[\frac{1}{2} + \frac{1}{4}\right]x_2(\tau_k)} \end{aligned}$$

Algoritmi asincroni - exemplu

$$P_1\colon\ x_1(t+1) = \frac{x_1(t)}{2} + \frac{x_2(\tau_k)}{2}, \qquad \tau_k \le t < \tau_{k+1}$$

Între momentele τ_k și τ_{k+1}, P_1 menține $x_2(\tau_k)$ constant și execută iterația de mai sus de $\tau_{k+1} - \tau_k$ ori:

$$x_1(\tau_k + i) = \left(\frac{1}{2}\right)^i x_1(\tau_k) + \left[\frac{1}{2} + \frac{1}{4} + \cdots + \left(\frac{1}{2}\right)^i\right] x_2(\tau_k)$$

Algoritmi asincroni - exemplu

$$P_1\colon\ x_1(t+1) = \frac{x_1(t)}{2} + \frac{x_2(\tau_k)}{2}, \qquad \tau_k \le t < \tau_{k+1}$$

Între momentele τ_k și τ_{k+1}, P_1 menține $x_2(\tau_k)$ constant și execută iterația de mai sus de $\tau_{k+1} - \tau_k$ ori:

$$x_1(\tau_{k+1}) = \left(\tfrac{1}{2}\right)^{\tau_{k+1}-\tau_k} x_1(\tau_k) + \left(1 - \left(\tfrac{1}{2}\right)^{\tau_{k+1}-\tau_k}\right) x_2(\tau_k)$$

Algoritmi asincroni - exemplu

$$P_1\colon\ x_1(\tau_{k+1}) = \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k} x_1(\tau_k) + \left(1 - \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k}\right) x_2(\tau_k)$$

$$P_2\colon\ x_2(\tau_{k+1}) = \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k} x_2(\tau_k) + \left(1 - \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k}\right) x_1(\tau_k)$$

Algoritmi asincroni - exemplu

$$P_1\colon\ x_1(\tau_{k+1}) = \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k} x_1(\tau_k) + \left(1 - \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k}\right) x_2(\tau_k)$$

$$P_2\colon\ x_2(\tau_{k+1}) = \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k} x_2(\tau_k) + \left(1 - \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k}\right) x_1(\tau_k)$$

Notăm $\epsilon_k = 2\left(\tfrac{1}{2}\right)^{\tau_{k+1}-\tau_k}$:

$$\begin{aligned} |x_2(\tau_{k+1}) - x_1(\tau_{k+1})| &\le (1-\epsilon_k)|x_2(\tau_k) - x_1(\tau_k)| \\ &\le \textstyle\prod_i (1-\epsilon_i)\,|x_2(0) - x_1(0)| \end{aligned}$$

Algoritmi asincroni - exemplu

$$P_1:\ x_1(\tau_{k+1}) = \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k} x_1(\tau_k) + \left(1 - \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k}\right) x_2(\tau_k)$$
$$P_2:\ x_2(\tau_{k+1}) = \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k} x_2(\tau_k) + \left(1 - \left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k}\right) x_1(\tau_k)$$

Notăm $\epsilon_k = 2\left(\frac{1}{2}\right)^{\tau_{k+1}-\tau_k}$:

$$|x_2(\tau_{k+1}) - x_1(\tau_{k+1})| \le (1-\epsilon_k)|x_2(\tau_k) - x_1(\tau_k)| \le \prod_i (1-\epsilon_i)\, |x_2(0) - x_1(0)|$$

Condiție de convergență:

$$\lim_{k\to\infty} \prod_{i=0}^{k} (1-\epsilon_i) = 0.$$

Sistem liniar pătratic

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

Este echivalent cu

$$x_1 = -\frac{1}{A_{11}}(A_{12}x_2 + \cdots + A_{1n}x_n - b_1)$$
$$x_2 = -\frac{1}{A_{22}}(A_{21}x_1 + \cdots + A_{1n}x_n - b_2)$$
$$\cdots$$
$$x_n = -\frac{1}{A_{nn}}(A_{n1}x_1 + \cdots + A_{nn-1}x_{n-1} - b_n)$$

Algoritmul Jacobi sincron

Ideea algoritmului Jacobi sincron:

$$P_i:\ x_i(t+1) = -\frac{1}{A_{ii}}\left(\sum_{j\ne i} A_{ij}x_j(t) - b_i\right)$$

$$x(t+1) = D^{-1}\big(b - Rx(t)\big)$$

- Pentru calcularea $x_i(t+1), P_i$ așteaptă $x_j(t)$ de la P_j, unde $j = 1, \dots, p, j \ne i$
- Este necesar ca fiecare procesor să stocheze $x(t)$!

Algoritmi asincroni

Algoritm Jacobi asincron*:*

$$x_i(t+1) = -\frac{1}{A_{ii}}\left(\sum_{j\ne i} A_{ij}x_j\left(\tau_j^i(t)\right) - b_i\right)$$

Sub ipotezele precedente, alg. Jacobi converge la $x^* = -D^{-1}Rx^* + D^{-1}b$.

În acest caz, ipotezele de convergență se reduc la:
- $D^{-1}R$ contracție în raport cu $\|\cdot\|_\infty$
- $raza\ spectrala$ a matricii $|D^{-1}R|$ sa fie subunitara, i.e. $\rho(|D^{-1}R|) < 1$.

Ipotezele sunt foarte restrictive pentru algoritmii de medie (consens).