AlgoDist : tout pour le contrôle

Algorithmique distribuée, Master 2 Ubinet / SI5 SSE, F. Baude. Contrôle papier du lundi 2 novembre 2026, 3 h. Version couleur. Fiches par chapitre, algorithmes en pseudo-code court, réponses types des annales, pièges, schémas. Cours en ligne : cours.lsro.fr/algodist

Sommaire

Le contrôle 1

Ce qui tombe aux examens 1

Quel algorithme quand 2

Chapitre 1 : Introduction et élection 3

Exercices corrigés 4

Chapitre 2 : Temps, coupures et snapshots 7

Exercices corrigés 8

Chapitre 3 : Tolérance aux fautes et recovery 10

Exercices corrigés 11

Chapitre 4 : Communications de groupe 13

Exercices corrigés 14

Chapitre 5 : Détecteurs de fautes et consensus 16

Exercices corrigés 17

Chapitre 6 : Transactions distribuées et état global 19

Exercices corrigés 20

Annexe : exclusion mutuelle 23

Exercices corrigés 24

Références et annales 25

Le contrôle

DATE Lundi 2 novembre 2026, après-midi, 3 h, examen papier.

DOCUMENTS Autorisé (message du prof du 1er novembre) : les slides imprimées distribuées, tous ses tests, et ses notes, qui peuvent être imprimées.

LE PROF Selon le prof : long mais facile. Surtout faire tourner des algorithmes fournis et les comprendre, plus des questions sur les slides (certaines vues en cours). Les exercices Moodle et leurs corrections sont utiles.

Ce qui tombe aux examens

D'après les annales (2011 à 2019 et 2022) : thèmes qui reviennent, avec les années et le chapitre de ce document.

Thème	Années	Chap.
Checkpointing : CIC (Briatico), calcul du N de recovery, forced checkpoints	2011, 2012-13, 2013-14	3
Checkpoint avant / après chaque envoi, graphe de dépendances, ligne de recovery, domino	2012, 2022	3
Ordres causal et total : 2 propriétés causales (safety / liveness), séquenceur sans FIFO, reprise du séquenceur	2011, 2012-13, 2013-14	4
CBCAST à horloges vectorielles : livrabilité, états hold	2017-18, 2019	4
Consensus uniforme contre non uniforme, flooding en N-1 étapes, hiérarchique	2011, 2012, 2019	5
Détecteurs ◇P, élection de leader, « pas plus d'un jour » = synchrone	2011, 2012	5
Exclusion mutuelle : Ricart-Agrawala, token ring, Suzuki-Kasami, preuves ME1 à ME3	2011, 2012-13, 2013-14, 2017-18	A
Élection : probe-echo (compter processus ou arêtes), Chang et Roberts, Bully, inondation, Franklin	2011, 2012, 2012-13, 2013-14, 2017-18, 2019	1
Temps : Cristian (preuve de la précision), horloges vectorielles (diagonale), estampilles	2012, 2017-18, 2013-14	2
Terminaison : quand détecter, Misra (nombre de tours), jeton sur anneau	2012-13, 2017-18, 2019	6

Quel algorithme quand

Situation	Algorithme	Chap.
Élire le plus grand sur un anneau à sens unique, sans panne	Chang et Roberts (fin détectée gratuitement, pire cas n(n+1)/2)	1
Élire sur un anneau à double sens avec garantie au pire cas	Franklin (2N log N messages au pire, log N rondes)	1
Collecter ou diffuser avec confirmation sur un graphe quelconque, un initiateur	Probe-Echo (2E messages, arbre couvrant)	1
Élire avec pannes de processus, graphe complet, ids connus, timeout	Bully ($O(n^2)$, pire $O(n^3)$)	1
Ordonner des événements, savoir si deux événements sont concurrents	Horloges vectorielles (Lamport seul ne prouve pas la concurrence)	2
Photo cohérente de l'état global (processus + canaux)	Chandy-Lamport (canaux FIFO, marqueurs)	2
Synchroniser une horloge sur un serveur de temps / entre pairs	Cristian (externe, ±(RTT/2 - min)) / Berkeley (interne, moyenne)	2
Reprendre après panne en revenant à une coupe cohérente	Checkpointing coordonné (bloquant ou non) ou CIC (index piggybacké)	3
Ne faire reculer que le processus fautif	Message logging pessimiste (log avant exécution)	3
Livrer à tous les corrects si un correct livre (avec détecteur P)	Reliable Broadcast lazy ; uniforme : URB All-Ack	4
Respecter la causalité des diffusions	CBCAST (vecteurs de diffusion, livrer si $VC_m \leq VC$)	4
Même ordre de livraison partout	Total order : séquenceur (2N messages) ou Lamport + FIFO (mysterious broadcast)	4
Consensus avec un détecteur P, décision de chaque chef à son tour	Consensus hiérarchique (non uniforme) ; uniforme : PROPOSAL + ACK	5
Consensus avec ◇S et majorité de corrects	Chandra-Toueg (coordinateur tournant, $f < n/2$) ; Paxos si leader seulement à terme	5
Exclusion mutuelle répartie avec horloges, canaux FIFO	Lamport (3(N-1) messages) ; sans FIFO et sans release : Ricart-Agrawala (2(N-1) messages)	A
Exclusion mutuelle avec un jeton	Anneau à jeton (1 message par sortie, (N-1)-fairness) ; graphe complet : Suzuki-Kasami (req[], last[], jeton = (Q, last))	A
Valider ou annuler une transaction répartie	2PC (bloquant) ; 3PC (non bloquant sous ses hypothèses)	6
Savoir que l'application est terminée (comptage)	Mattern, 4 compteurs : deux vagues, E1=R1=E2=R2	6
Savoir que l'application est terminée (jeton sur anneau)	Misra (FIFO, token = n) ; Dijkstra-Feijen-van Gasteren (blanc ou noir) ; sans n ni anneau : Dijkstra-Scholten	6
Interblocage : ressources (ET) / communication (OU)	Chandy-Misra-Haas : sondes P(i,s,r) / probe-echo avec acks	6

Chapitre 1 | Introduction et élection

TOMBE Tombe à presque chaque examen : probe-echo (compter processus ou arêtes, 2011 et 2012), Chang et Roberts avec tous candidats (2012-13), Bully et connaissance de N (2013-14), inondation en D rondes (2017), Franklin (2019).

Explications

Pourquoi un modèle et des actions gardées

Avant tout algorithme il faut dire ce que le système permet : des processus qui ne partagent rien et n'ont pas d'horloge commune, qui ne se parlent que par messages. Une **action gardée** est une règle « si la garde est vraie, exécute le corps d'un coup » : **Ip** quand la garde porte sur l'état local, **Rp** quand elle porte sur l'arrivée d'un message. Le non-déterminisme (plusieurs gardes vraies) force à raisonner pour *tous* les ordres possibles, pas pour un seul.

Conséquence pratique à l'examen : quand on déroule un algorithme, on est libre de choisir l'ordre des actions, mais il faut que le résultat soit juste quel que soit cet ordre. Le piège classique est un message qui arrive avant l'action interne de démarrage : sans garde *started* dans les deux règles, l'action interne reste bloquée.

Chang et Roberts, pas à pas

Chaque candidat écrit son numéro sur un papier et le passe à son voisin. Un processus qui reçoit un papier plus grand que tout ce qu'il a vu le fait suivre, sinon il le jette. Un papier qui revient à son auteur a battu tout le monde : l'auteur est le plus grand, il l'a appris sans rien demander (fin détectée gratuitement) et proclame.

Le coût est la somme des distances parcourues par les papiers. Un papier meurt dès qu'il rencontre un processus qui a vu plus grand. Si les ids décroissent dans le sens du tour, chaque papier survit longtemps (pire cas) ; s'ils croissent, tous meurent au premier saut sauf le max (meilleur cas).

Probe-Echo : un arbre qui se construit tout seul

L'initiateur envoie un probe à tous ses voisins. Un processus qui reçoit son premier probe retient l'expéditeur comme **parent** et envoie des probes aux autres voisins. Quand il a reçu autant de messages (probe ou echo) qu'il a de voisins, il envoie un echo à son parent. L'initiateur termine quand il a tout reçu.

Le résultat est un arbre couvrant qui dépend des vitesses des liens. Chaque arête porte exactement deux messages : probe et echo si elle est dans l'arbre, deux probes croisés sinon. D'où le total 2E.

Quel algorithme d'élection choisir

Anneau à sens unique sans panne : Chang et Roberts. Anneau à deux sens avec garantie au pire cas : Franklin (la moitié des actifs disparaît à chaque ronde). Graphe quelconque sans panne : Tel (un Probe-Echo par candidat, la vague du max l'emporte). Pannes de processus avec délai borné : Bully (le plus grand vivant gagne, mais deux leaders sont possibles si un processus redémarre). Serveur fiable : serveur central (le premier candidat arrivé gagne).

À noter sur la feuille

Modèle

Processus P0..Pn, **mémoire locale**, ni mémoire partagée ni horloge globale. Envoi seulement aux **voisins**. Canaux fiables, FIFO. Pas de panne sauf mention.

Temps : **async** (défaut, délai fini non borné : lent et mort indistinguables, pas de timeout fiable), **sync** (borne Δ connue, rondes, timeout 2Δ fiable), async borné connu (timeout possible, Bully).

FIFO : sur un canal P→Q, réception dans l'ordre d'envoi. Rien de garanti entre deux canaux différents. Fabriquer FIFO : numéros de séquence et attente.

Actions gardées

Ip: { garde } règle interne (ne reçoit jamais de message). **Rp: { msg }** règle message (receive au début). Plusieurs gardes vraies : choix au hasard, action exécutée **en entier** (atomique).

Piège : un message traité avant Ip rend $M \neq 0$ et Ip reste bloquée à jamais. Correctif : **started** dans les deux gardes ; une garde fausse laisse le message dans la boîte.

Coût

Coût = **messages** (compter les flèches) + **temps** (plus longue chaîne de messages qui se déclenchent, 1 unité = 1 message). Messages envoyés en parallèle = **1 seule** unité de temps.

Toujours donner le nombre ET la situation qui le provoque (pire, meilleur cas). $O(1) < O(n) < O(n \log n) < O(n^2) < O(n^3)$.

EN *Cost is counted in messages and in time, where one time unit is one message in a causal chain.*

Élection

Casser la symétrie : choisir **un** coordinateur ; un seul élu et tous les non fautifs connaissent le même. Valeurs uniques, ordre total (souvent max id). Étapes : détection, préparation, élection, proclamation.

Approximation distribuée : chacun garde le meilleur vu et l'améliore. Complet naïf : pire $O(N^3)$ (reçu en ordre croissant), meilleur $O(N^2)$, temps N-1. Complet avec N connu : N(N-1) messages, temps 1.

EN *Election breaks symmetry: exactly one leader, and every correct process knows the same one.*

Chang et Roberts

Anneau à **sens unique**, pas de panne. Je passe un numéro seulement s'il est > M (le plus grand vu), sinon je le jette. Je reçois **mon propre numéro** : j'ai gagné (fin détectée gratuitement). Proclamation : fait le tour (+ n messages).

Coût = somme des distances parcourues par chaque papier. **Pire** : ordre décroissant dans le sens du tour, le papier i fait i sauts, $n(n+1)/2 = O(n^2)$. **Meilleur** : ordre croissant, (n-1) + n = 2n-1. Moyenne O(n log n). Temps O(n). Plus n messages de proclamation. 2 candidats sur N : N + d ($1 \leq d \leq N-1$), meilleur N+1, pire 2N-1, moyenne 3N/2.

V1 : un dormeur qui reçoit j le relaie et ne candidate jamais, élu = max des réveillés seuls. V2 : un dormeur qui reçoit j < moi jette j et envoie MON numéro, élu = max global.

EN *A process forwards an id only if it is larger than the largest it has seen; receiving its own id means it won.*

Probe-Echo

1 initiateur, réseau quelconque connexe, liens bidirectionnels. 1er probe reçu de q : **parent := q**, probe aux autres voisins. Chaque message reçu : compteur++ ; compteur = nombre de voisins : **echo** au parent (initiateur : fin).

Construit un **arbre couvrant**. Messages **2E** (arête de l'arbre : probe + echo ; hors arbre : 2 probes croisés). Temps ≤ 2N. Usages : diffusion avec confirmation, collecte dans l'echo (sac S, union). Un seul echo par processus. Compter les arêtes : echo 0 si déjà visité (2012).

L'arbre dépend des vitesses des liens (parent = probe arrivé en premier) : triangle depuis A, 3 arbres possibles.

EN *The first probe received defines the parent; a process answers with an echo once every neighbour has answered.*

Élection de Tel, Franklin, Bully, serveur central

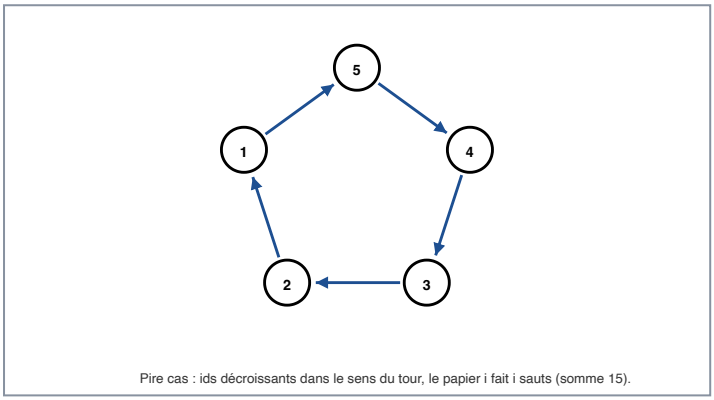
Tel (réseau quelconque) : chaque candidat lance un Probe-Echo ; j > M : changer de camp (M := j, parent := q, nb := 1, relayer) ; j = M : nb++ ; j < M : ignorer. Seule la vague du max revient complète. ≤ 2E par vague, pire O(N·E).

Franklin : anneau double sens, pas de panne, rondes. Un actif envoie son numéro à gauche et à droite, compare aux 2 actifs voisins : un voisin plus grand, éliminé (relais seulement) ; reçoit son propre numéro : élu. Au moins la moitié éliminée par ronde : ≤ log N rondes. 2N messages par ronde, 2N log N au pire, garanti au pire (≠ C&R).

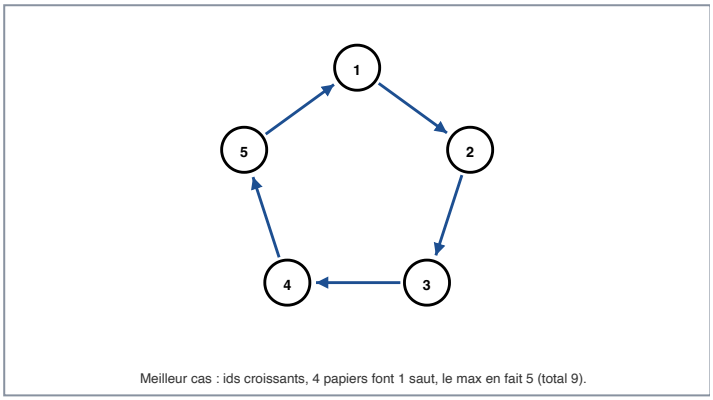
Bully : pannes possibles, liens fiables, délai borné, graphe complet, ids connus. Leader ⇔ tous les plus grands morts. Pi envoie Election aux j > i ; Pj vivant répond Reply et devient candidat ; personne ne répond avant timeout : Leader aux j < i. O(n^2) messages, O(n^3) au pire (n-1 pannes pendant l'élection), temps O(n). Piège slide 27 : un processus redémarré ou lent se proclame aussi, deux leaders.

Serveur central : étoile, serveur jamais en panne, oracle de panne du leader. 1er be_candidate = élu (pas forcément le max), search_leader servi seulement si leader ≠ vide. 2N messages, temps O(1).

Schémas



Chang et Roberts, pire cas.



Chang et Roberts, meilleur cas.

Algorithmes clés

Chang et Roberts (anneau à sens unique)

```
Ip: { ¬started }  started := vrai ; M := p ; send ⟨p⟩ au suivant
Rp: { message ⟨j⟩ arrivé }
    si j = p  alors leader := p ; proclamer
    sinon si j > M  alors M := j ; send ⟨j⟩ au suivant
    sinon jeter j
```

Probe-Echo

```
Rp: { ⟨probe⟩ de q }  si premier probe : parent := q ; probe aux voisins ≠ q
Rp: { ⟨probe⟩ ou ⟨echo⟩ reçu }  compteur++
    si compteur = |voisins| : echo à parent  // initiateur : fin
```

Bully (Pi)

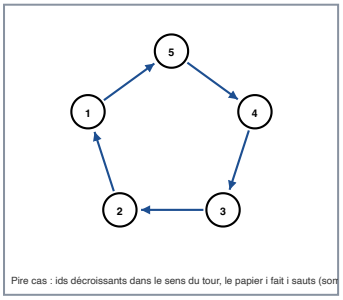
```
Ip: { détection du leader mort }  send ⟨Election⟩ à tous les j > i ; attendre timeout
Rp: { ⟨Election⟩ de k < i }  send ⟨Reply⟩ à k ; lancer sa propre élection
Rp: { ⟨Reply⟩ }  attendre ⟨Leader⟩  // timeout sans Leader : recommencer
timeout sans Reply : leader := i ; send ⟨Leader⟩ à tous les j < i
```

Exercices corrigés (chapitre 1)

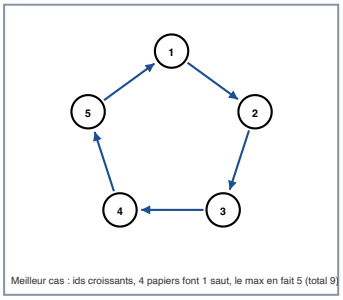
Chang et Roberts : meilleur et pire cas (n = 5)

annales 2012-13, cours

Énoncé. Anneau de 5 processus, sens du tour = sens des flèches, tous candidats. Compter les messages d'élection pour des ids croissants puis décroissants dans le sens du tour.



Ids 5,4,3,2,1 dans le sens du tour.



Ids 1,2,3,4,5 dans le sens du tour.

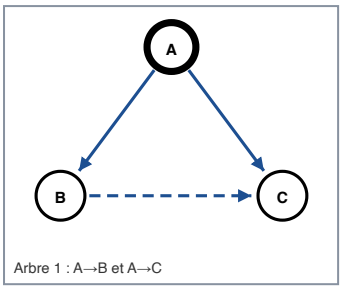
1. **Décroissant (5,4,3,2,1)** : le papier 1 est jeté par le voisin (5) après 1 saut ; le papier 2 passe chez 1 (2 > 1) puis est jeté par 5 : 2 sauts ; le papier i fait i sauts, le papier 5 fait le tour complet (5 sauts). Total 5+4+3+2+1 = **n(n+1)/2 = 15** = O(n^2).
2. **Croissant (1,2,3,4,5)** : les papiers 1 à 4 sont jetés par leur voisin plus grand, 1 saut chacun (4 messages) ; le papier 5 fait le tour (5 sauts). Total 4+5 = **2n-1 = 9**.
3. Ajouter la proclamation : n messages de plus dans les deux cas. Temps O(n). Moyenne O(n log n).

A ECRIRE Pire cas n(n+1)/2 messages (ids décroissants dans le sens du tour), meilleur cas 2n-1 (croissants), plus n pour la proclamation.

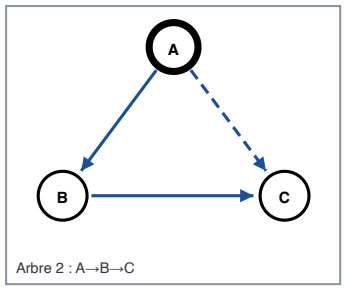
EN *Worst case n(n+1)/2 election messages, best case 2n-1, plus n messages to announce the leader.*

Probe-Echo sur un triangle cours (test)

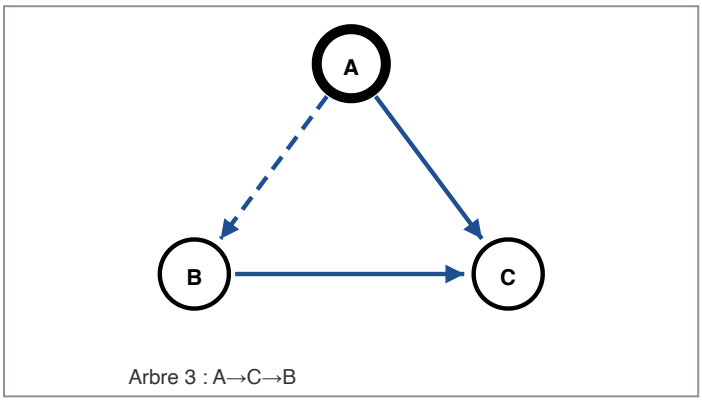
Énoncé. Triangle A, B, C, initiateur A. Quels arbres peut-on obtenir, combien de messages ?



Arbre 1



Arbre 2



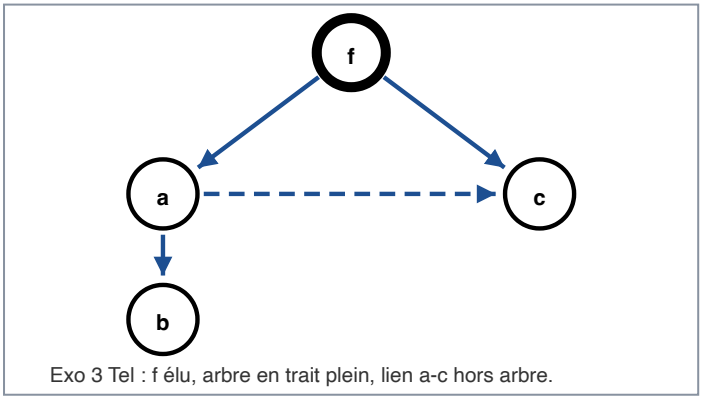
Arbre 3

1. A envoie un probe à B et à C. Le **premier probe reçu** fixe le parent : si B et C reçoivent chacun le probe de A d'abord, arbre 1. Si B reçoit A puis transmet à C avant que le probe de A n'arrive, C a pour parent B (arbre 2), et symétriquement arbre 3.
2. Arête d'arbre : un probe et un echo (sens opposés). Arête hors arbre : deux probes qui se croisent. Chaque arête porte donc 2 messages : **2E = 6**.
3. Temps ≤ 2N.

> A ECRIRE Trois arbres possibles selon les vitesses des liens ; 2E = 6 messages.
> EN *The tree depends on which probe arrives first; every edge carries two messages, so 2E in total.*

Élection de Tel (exo 3 : a, b, c, f) cours

Énoncé. Réseau a, b, c, f avec les liens f-a, f-c, a-b, a-c. Appliquer Tel avec tous candidats.



Arbre de la vague de f.

1. Chaque candidat lance un Probe-Echo. Un processus qui reçoit une vague d'un candidat plus grand que M change de camp (M := j, parent := q, nb := 1) et relaie ; un candidat égal à M : nb++ ; plus petit : ignoré.
2. Seule la vague du plus grand revient complète. Ici f est le plus grand : arbre f→a, f→c, a→b ; le lien a-c n'est pas dans l'arbre (deux probes croisés).
3. Coût ≤ 2E par vague, pire O(N·E).

> A ECRIRE f est élu ; arbre f→a, f→c, a→b, lien a-c hors arbre.
> EN *Only the wave of the highest id completes, so its initiator is elected.*

Routage par approximation (exo 2 du TD) cours

Énoncé. Graphe P0..P4 avec liens P0-P1 (1), P1-P2 (2), P0-P4 (5), P1-P3 (3), P3-P4 (6), P2-P4 (1). Évolution de la table de P0.

1. Chaque nœud garde (distance, via) pour chaque destination, au départ seulement ses liens directs. Il envoie sa table à ses voisins ; à la réception de la table de v par un lien de coût c : si c + d_v[x] < d[x], il met à jour via v ; si sa table change, il la renvoie.
2.

moment	vers P1	vers P2	vers P3	vers P4
départ	1	?	?	5
ronde 1	1	3 (via P1)	4 (via P1)	5
ronde 2	1	3	4	4 (via P1) : 1+3 < 5
ronde 3	1	3	4	4 : stable

3. La valeur directe 5 vers P4 est d'abord crue puis améliorée : c'est l'**approximation** (on part d'une valeur fausse et on l'améliore jusqu'à ce que plus rien ne bouge).

> A ECRIRE P0 final : P1 1, P2 3 via P1, P3 4 via P1, P4 4 via P1.
> EN *Each node improves its estimate whenever a neighbour offers a shorter path, until no table changes.*

Test 1 : message avant l'action interne, capteurs, anonymes cours (tests)

Énoncé. Test 1 Q1-Q2 ; exo 4 capteurs (température max) ; exo 5 numéros uniques pour processus anonymes.

1. **Q1** : un message traité avant l'action interne fait M ≠ 0 donc la garde de Ip ne sera plus vraie : Ip bloquée pour toujours. **Q2** : mettre *started* dans les deux gardes ; une garde fausse laisse le message dans la boîte.
2. **Capteurs** : m := ma température, je diffuse ; entendu v > m : m := v et je rediffuse ; sinon rien. Le max atteint tout le monde (réseau connexe). Fin : détection de terminaison, ou D diffusions si le diamètre est connu.
3. **Anonymes (graphe complet)** : par rondes, chaque processus sans numéro tire pile ou face, ajoute le bit à sa suite et l'envoie à tous ; une suite unique devient son numéro, les autres retirent. Fin avec probabilité 1, environ log N rondes.

> A ECRIRE started dans les deux gardes ; diffusion du max avec rediffusion si plus grand ; tirage de bits par rondes.

Réponses types des annales

Q. Chang et Roberts : meilleur et pire cas en messages ?

R. Meilleur : ids croissants dans le sens du tour, 2n-1 messages d'élection (+ n de proclamation). Pire : ids décroissants dans le sens du tour, le papier i fait i sauts, n(n+1)/2.

> EN *Best case: ids increasing along the ring, 2n-1 election messages. Worst case: decreasing, n(n+1)/2.*

Q. Pourquoi Franklin est-il meilleur que C&R au pire cas ?

R. À chaque ronde au moins la moitié des actifs est éliminée (de deux actifs voisins, le plus petit tombe) : au plus log N rondes de 2N messages, donc 2N log N garanti.

EN *At least half of the active processes are eliminated each round, so there are at most log N rounds of 2N messages.*

Q. Probe-Echo : combien de messages et pourquoi ?

R. 2E : chaque arête de l'arbre porte un probe et un echo, chaque arête hors arbre porte deux probes croisés.

EN *Every edge carries exactly two messages, so the total is 2E.*

Q. Bully : quel problème si un processus redémarre ?

R. Un processus de plus grand id redémarré (ou lent, timeout mal calibré) se proclame leader même si un autre vient d'être élu : deux leaders, vue incohérente.

EN *A recovering process with the highest id proclaims itself leader, so two leaders may coexist.*

PIEGES **à éviter**

- Compter le temps en nombre de messages d'une chaîne causale, pas en nombre total de messages.
- C&R : la proclamation ajoute n messages, ne pas les oublier dans le total.
- Élection : un seul élu et tous d'accord, pas forcément le plus grand id.
- Async : impossible de savoir si un autre candidat va arriver (serveur central : le premier candidat suffit).
- Bully exige des ids connus et un timeout, donc un délai borné.

Chapitre 2 | Temps, coupures et snapshots

TOMBE Cristian : preuve de la précision (2012), délai d chez le serveur (2017) ; horloges vectorielles : le max des vecteurs d'événements concurrents est la diagonale (2013-14) ; estampilles et ordre de livraison ; Chandy-Lamport sans FIFO.

Explications

Causalité : ce qu'on peut savoir sans horloge

Sans horloge globale, on ne peut pas dire « a a eu lieu avant b » au sens du temps réel. On définit à la place **a → b** : une information a pu voyager de a à b (même processus dans l'ordre, envoi puis réception, transitivité). Deux événements non reliés sont **concurrents**. Ce n'est pas un ordre total : plusieurs « passés » sont compatibles avec ce qu'on observe.

Lamport contre vecteurs

L'horloge de Lamport est un entier qui respecte la causalité dans un sens : a → b implique L(a) < L(b). L'inverse est faux : deux événements concurrents ont des valeurs comparables quand même. Elle sert à fabriquer un ordre total (L, id) cohérent avec la causalité.

L'horloge vectorielle a une case par processus. Elle caractérise exactement la causalité : V(a) < V(b) si et seulement si a → b, et deux vecteurs incomparables signifient concurrence. Prix : N entiers par message.

Coupes et snapshots

Une **coupe** choisit un point par processus. Elle est **cohérente** si aucun message n'est reçu dans le passé de la coupe et envoyé dans son futur (orphelin). Les messages envoyés avant et reçus après sont simplement **en transit**. Chandy-Lamport prend une telle photo en ligne : le marqueur, envoyé sur chaque canal sortant juste après la photo locale, sépare les messages « avant » des messages « après ». Il faut des canaux FIFO pour que cette séparation tienne.

Synchroniser les horloges physiques

Cristian : un client interroge un serveur de temps et corrige avec la moitié du RTT ; l'incertitude vient de l'asymétrie inconnue des deux trajets. Berkeley : pas de serveur de référence, un maître fait la moyenne des horloges (en écartant les valeurs aberrantes) et renvoie à chacun sa correction.

À noter sur la feuille

Causalité

e1 → e2 (happened-before) : une information a pu aller de e1 à e2. Trois règles : même processus et avant ; envoi puis réception ; transitivité.

Ni e1 → e2 ni e2 → e1 : **e1 ‖ e2** (concurrents). Tester : chercher un chemin. ‖ n'est pas transitif.

EN *e1 happened before e2 if a piece of information could have travelled from e1 to e2.*

Horloges de Lamport

+1 avant chaque événement ; l'envoi porte L ; réception de l : **L := max(L, l) + 1**.

a → b ⇒ L(a) < L(b), **pas la réciproque**. Ordre total (L, id). L(e) = plus longue chaîne causale qui mène à e.

EN *Lamport clocks respect causality, but L(a) < L(b) does not imply a happened before b.*

Horloges vectorielles

V[i]++ avant chaque événement ; l'envoi porte V ; réception : **max case par case** puis V[i]++. Défaut : N cases.

V1 < V2 ⇔ e1 → e2. Un < et un > : e1 ‖ e2. Deux événements concurrents : le max des vecteurs d'un trio d'événements 2 à 2 concurrents donne la **diagonale** (Vt1[1], ..., Vtn[n]).

EN *V(a) < V(b) holds if and only if a happened before b; two incomparable vectors mean concurrency.*

Coupes cohérentes

Coupe cohérente ⇔ **pas d'orphelin** (envoi dans le futur, réception dans le passé) ⇔ VCut[k] = Vk[k]. S = R ne suffit pas (un envoi non vu et une réception non vue se compensent).

Effacer une réception = message en transit (sans danger). Effacer un envoi déjà reçu = orphelin.

Chandy-Lamport

≠ horloges de Lamport : pas d'horloge, but = **photo de l'état global** (état de chaque processus + messages en transit). FIFO, pas de perte, pas de panne, tout processus peut lancer.

Initiateur : photo + marqueur sur chaque canal sortant + écoute tous les entrants. 1er marqueur reçu sur c : photo + c = vide + marqueur sur chaque sortant + écoute les autres. Message normal sur canal écouté : noté (en transit). Marqueur suivant sur c : état(c) = messages notés depuis la photo. Fin : marqueur reçu sur tous les entrants.

Banque : P=70, Q=90, R=100, 10 en route P→Q et 30 Q→P, R lance : photos R=100, Q=90, P=70, canaux P→Q {10}, Q→P {30}, total 300. Naïf (slide 17) : B photo puis envoie M à C, C reçoit puis photo : 310 au lieu de 300.

EN *The marker separates, on each channel, messages sent before the sender's snapshot from those sent after; FIFO guarantees it.*

Sans FIFO (exo 5) et exo 6

P et Q à 100 (total 200), canal P→Q non FIFO. Cas 1 : P envoie m (10), photo P=90, marqueur ; le marqueur double m : Q photo 100, canal vide, m arrive après, noté nulle part : **190** (perte). Cas 2 : P photo 100, marqueur, puis m' (10) ; m' double le marqueur : Q=110 puis photo : réception sans envoi, orphelin, **210**.

Exo 6 (anneau P⇄Q, un jeton m, état = nombre de réceptions) : Sp=101, Sq=102, P→Q vide ; Q→P vide si Q garde m, {m} si Q l'a renvoyé avant le marqueur. Un seul jeton, cohérent.

EN *Without FIFO an application message can overtake the marker, so it is recorded nowhere or received without its send.*

Cristian et Berkeley

Cristian (synchro externe, serveur S) : P demande à t0, S répond t, P reçoit à t1. RTT = t1 - t0. P règle à **t + RTT/2** ; répéter, garder le RTT minimal. S a lu t dans [t0 + min, t1 - min] : précision **±(RTT/2 - min)** (min inconnu : ±RTT/2). Exemple : RTT 20 ms, min 4 ms : ±6 ms.

Berkeley (synchro interne) : le maître interroge tous, corrige par RTT/2, fait la moyenne en ignorant les écarts > δ, renvoie à chacun son ajustement. Exemple : maître 3:00, A 3:25, B 2:50 : moyenne 3:05, ajustements +5, -20, +15.

EN *Accuracy is ±(RTT/2 - min), since the server's reading lies between t0 + min and t1 - min.*

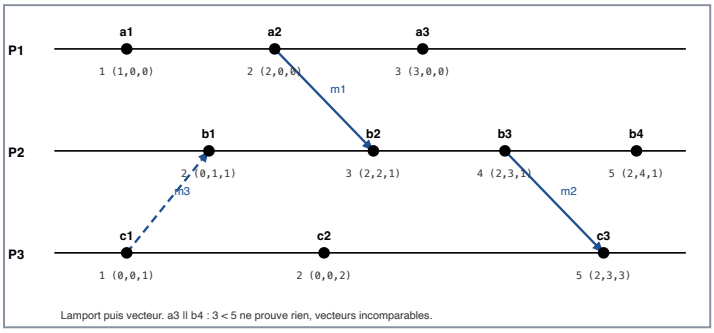
Exos 3 et 4 (estampilles)

Exo 3 (2 processus, ressource exclusive, plus petite estampille gagne, égalité : id) : pour redemander il faut avoir libéré donc échangé des messages, l'écart entre requêtes ≤ 1. Compter **modulo 3** : x plus ancien que y ⇔ y = x + 1 (mod 3). Modulo 2 insuffisant. Règle : 2 × écart + 1 valeurs.

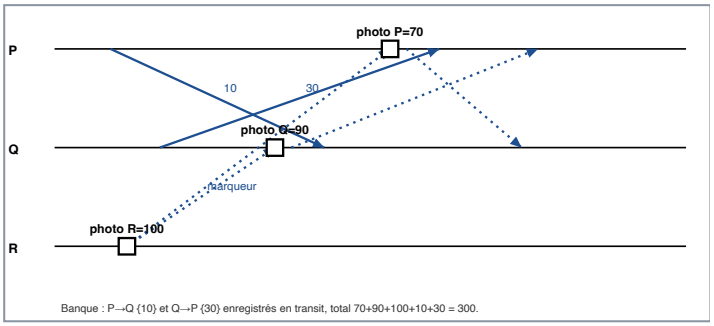
Exo 4 (N > 2, FIFO, envois infinis, traiter par estampille croissante) : last[k] = dernière estampille reçue de k. Traiter M (de j) ssi pour tout k ≠ j, last[k] > ts(M) ; sinon le garder. Un envoi de données à un processus : **message nul** de même estampille à tous les autres. Vivacité grâce à « envois infinis ».

EN *A process delivers M only once it has received from every other process a message, possibly null, with a larger timestamp.*

Schémas



Horloges de Lamport et vectorielles (diagramme du cours).



Chandy-Lamport, exemple bancaire P, Q, R.

Algorithmes clés

Horloge de Lamport

```
événement interne : L := L + 1
envoi : L := L + 1 ; send ⟨m, L⟩
réception ⟨m, l⟩ : L := max(L, l) + 1
```

Horloge vectorielle (Pi)

```
tout événement : V[i] := V[i] + 1
envoi : send ⟨m, V⟩
réception ⟨m, W⟩ : pour tout k : V[k] := max(V[k], W[k])
; V[i] := V[i] + 1
```

Chandy-Lamport (Pi)

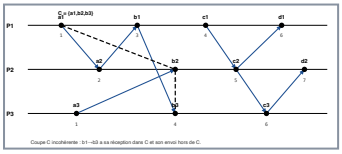
```
Ip: { initiateur }  photo ; marqueur sur chaque sortant
; écouter tous les entrants
Rp: { ⟨marqueur⟩ sur c, premier }  photo ; état(c) :=
vide ; marqueur sur chaque sortant ; écouter les autres
Rp: { ⟨marqueur⟩ sur c, suivant }  état(c) := messages
notés depuis la photo ; fin si tous les entrants clos
Rp: { message normal sur c écouté }  noter (en transit)
```

Exercices corrigés (chapitre 2)

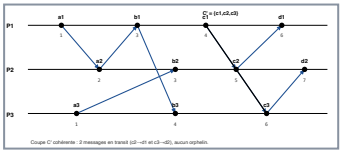
Exo 2 du TD : Lamport, vecteurs et coupes

TD2, valeurs vérifiées

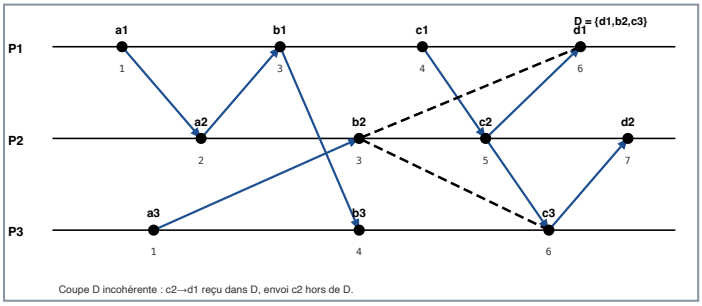
Énoncé. Trois processus, huit messages (a1→a2, a2→b1, a3→b2, b1→b3, c1→c2, c2→c3, c2→d1, c3→d2). Donner les horloges, puis dire quelles coupes sont cohérentes.



C incohérente.



C' cohérente.



D incohérente.

1. **Lamport** : P1 : 1, 3, 4, 6 ; P2 : 2, 3, 5, 7 ; P3 : 1, 4, 6. Exemple : b1 reçoit a2 (L=2) donc max(1,2)+1 = 3.
2. **Vecteurs** : a1 (1,0,0), b1 (2,1,0), c1 (3,1,0), d1 (4,3,1) ; a2 (1,1,0), b2 (1,2,1), c2 (3,3,1), d2 (3,4,3) ; a3 (0,0,1), b3 (2,1,2), c3 (3,3,3).
3. **Coupes** : C {a1,b2,b3} **incohérente** (b1→b3 : réception dans C, envoi b1 hors de C). C' {c1,c2,c3} **cohérente** (8 envois, 6 réceptions : 2 messages en transit). C'' {d1,d2,c3} **cohérente** (8 envois, 8 réceptions). D {d1,b2,c3} **incohérente** : le test par vecteur échoue, VD[2] = 3 ≠ b2[2] = 2.
4. **Q9** : trois événements 2 à 2 concurrents {c1,b2,b3} : le max des vecteurs est (3,2,2), c'est la diagonale (c1[1], b2[2], b3[3]).

A ECRIRE Une coupe est cohérente ssi VCut[k] = Vk[k] pour tout k, c'est-à-dire sans orphelin ; les messages en transit sont permis.

EN *The cut is consistent because no message is received before the cut and sent after it.*

Exo 1 du TD : P, Q, R et ses questions

TD2, valeurs vérifiées

Énoncé. Diagramme P, Q, R, messages p1→q3, p2→q6, q2→p5, q4→p3, q5→r2, q2→r1, q7→r5, r3→q9, q10→r6.

1. Le canal Q→P est **non FIFO** : q2→p5 double q4→p3 (envoyé avant, reçu après).
2. Lamport : p6 = 10, q11 = 12, r6 = 12. Contre-exemple : L(r1) = 3 < L(q6) = 7 sans r1 → q6.
3. Concurrents de q6 : p2', p3, p3', p4, p5, p6, r1, r2, r3, r4.
4. Q3 : ajouter q5→p2' est **possible** (pas de cycle causal), ajouter p3'→q3' est **impossible** (q3' → q4 → p3 → p3' créerait un cycle).

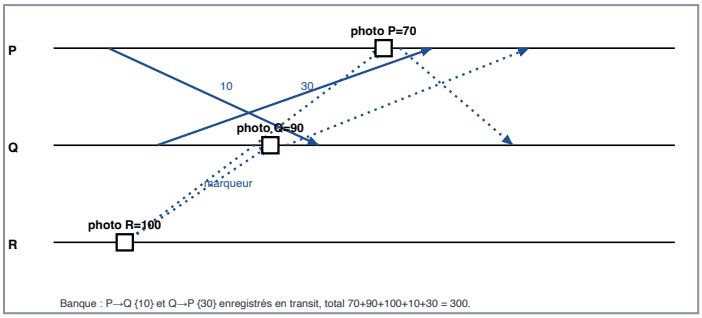
A ECRIRE Un message ajouté est possible s'il ne crée pas de cycle causal.

EN *Adding the message is impossible because it would create a causal cycle.*

Chandy-Lamport : exemple bancaire, exos 5 et 6

cours et TD2

Énoncé. P = 70, Q = 90, R = 100, 10 en route de P vers Q et 30 de Q vers P, R lance. Puis contre-exemple sans FIFO (exo 5) et anneau avec un jeton (exo 6).



Photos et canaux.

1. R prend sa photo (100) et envoie un marqueur sur chaque canal sortant. Q reçoit le marqueur : photo (90), canal R→Q vide, marqueurs sortants ; il enregistre les messages qui arrivent ensuite sur ses autres entrants. P de même (70).
2. Le message 10 (P→Q) arrive chez Q après sa photo et avant le marqueur de P : il est noté dans l'état de P→Q. Idem 30 pour Q→P. Total : 70+90+100+10+30 = **300**. La version naïve (slide 17) donnait 310.
3. **Exo 5, sans FIFO** : P et Q à 100. Cas 1 : P envoie m (10), photo P = 90, marqueur ; le marqueur double m, Q photographie 100 et enregistre le canal vide, m n'est compté nulle part : 190. Cas 2 : P photo 100, marqueur, puis m' ; m' double le marqueur : Q = 110 puis photo, réception sans envoi : 210.
4. **Exo 6** : Sp = 101, Sq = 102, canal P→Q vide ; Q→P vide si Q garde m, {m} si Q l'a renvoyé avant le marqueur : un seul jeton, cohérent.

A ECRIRE Le marqueur sépare sur chaque canal les messages envoyés avant la photo de ceux envoyés après ; FIFO le garantit.
EN *Without FIFO a message can overtake the marker, so the recorded state is inconsistent.*

Cristian : preuve de la précision (2012) et délai d (2017)

annales

Énoncé. Prouver la précision ±(RTT/2 - min). Puis : liens symétriques, serveur qui attend d avant de répondre.

1. P envoie à t0, S lit son horloge, répond, P reçoit à t1. La lecture de S a eu lieu après t0 + min (aller minimal) et avant t1 - min (retour minimal), donc dans un intervalle de longueur RTT - 2·min. P règle à t + RTT/2, au milieu : erreur au plus **RTT/2 - min**.
2. Exemple : RTT 20 ms, min 4 ms : ±6 ms.
3. **Liens symétriques** : chaque trajet vaut RTT/2, la déviation est nulle. **Avec délai d** chez S après lecture de l'horloge : le temps vrai à la réception est t + l + d, P règle à t + l + d/2 : P est en **retard de d/2**.

A ECRIRE Précision ±(RTT/2 - min) ; avec liens symétriques la déviation est nulle ; un délai d chez le serveur décale P de d/2.
EN *The error is bounded by RTT/2 - min because the server read its clock within [t0 + min, t1 - min].*

Berkeley, exos 3 et 4

TD2

Énoncé. Berkeley : maître 3:00, A 3:25, B 2:50. Exo 3 : horloges bornées. Exo 4 : livraison par estampille.

1. **Berkeley** : moyenne (3:00 + 3:25 + 2:50)/3 = 3:05 ; ajustements : maître +5, A -20, B +15. Les écarts supérieurs à δ sont ignorés dans la moyenne.
2. **Exo 3** : pour redemander la ressource il faut l'avoir libérée, donc avoir échangé des messages : les deux estampilles en compétition diffèrent d'au plus 1. Compter modulo 3 : x est plus ancien que y ssi y = x + 1 (mod 3) ; l'égalité se départage par l'id. Modulo 2 ne suffit pas (+1 et -1 se confondent).
3. **Exo 4** : pour livrer par estampille croissante, un processus garde last[k] (dernière estampille reçue de k, FIFO garantit que k n'enverra que plus grand). Il traite M (de j) ssi pour tout k ≠ j, last[k] > ts(M). Pour débloquer les autres, chaque envoi de données s'accompagne d'un **message nul** de même estampille aux autres. La vivacité vient de « envois infinis ».

A ECRIRE Moyenne puis ajustement par processus ; compteur modulo 3 ; livraison quand toutes les dernières estampilles reçues dépassent celle du message.

Réponses types des annales

Q. Cristian : prouver la précision ±(RTT/2 - min)

R. S a lu t après t0 + min (temps minimal d'aller) et avant t1 - min (temps minimal de retour). P règle à t + RTT/2 ; l'erreur est donc bornée par (RTT - 2·min)/2 = RTT/2 - min dans les deux sens.

EN *The server read the clock between t0 + min and t1 - min, so setting the clock to t + RTT/2 gives an error of at most RTT/2 - min.*

Q. Montrer par un exemple que Chandy-Lamport est faux sans FIFO

R. Deux processus à 100 (total 200). Le marqueur double m : Q photographie 100, le canal est enregistré vide, m arrive après et n'est compté nulle part, total 190. Ou m' double le marqueur : réception sans envoi, 210.

EN *Without FIFO an application message can overtake the marker, so the recorded cut is inconsistent.*

Q. Les horloges de Lamport suffisent-elles à détecter la concurrence ?

R. Non : a → b implique L(a) < L(b) mais pas la réciproque (contre-exemple L(a3)=3 < L(b4)=5 avec a3 || b4). Les horloges vectorielles donnent l'équivalence.

EN *No: L(a) < L(b) does not imply a happened before b. Vector clocks characterise causality exactly.*

Q. Quand une coupe est-elle cohérente ?

R. Quand aucun message n'a son envoi dans le futur et sa réception dans le passé de la coupe (pas d'orphelin). Avec les vecteurs : VCut[k] = Vk[k] pour tout k. Les messages en transit sont autorisés.

EN *A cut is consistent if no message is received before the cut and sent after it.*

PIEGES **à éviter**

- Chandy-Lamport n'est pas une horloge : il ne marche qu'avec des canaux FIFO.
- L(a) < L(b) ne prouve ni a → b ni l'inverse : utiliser les vecteurs pour la concurrence.
- Vecteur à la réception : max case par case puis V[i]++, pas l'inverse.
- Une coupe S = R n'est pas forcément cohérente.
- Cristian : l'erreur est symétrique, ±RTT/2 si min inconnu.

Chapitre 3 | Tolérance aux fautes et recovery

TOMBE À chaque examen : CIC (2011, 2012-13, 2013-14), checkpoint avant / après chaque envoi avec graphe de dépendances et ligne de recovery (2012, 2022), domino.

Explications

Deux façons de survivre à une panne

Checkpointing : on sauvegarde de temps en temps l'état ; après une panne, (presque) tous les processus reviennent à une coupe cohérente puis repartent. **Message logging** : on journalise les événements non déterministes (les réceptions) ; seul le fautif revient en arrière et rejoue la même exécution grâce au journal.

Pourquoi le non coordonné donne le domino

Si chacun checkpointe quand il veut, la dernière photo de chaque processus n'est pas forcément cohérente. En reculant un processus on efface un envoi que l'autre a déjà reçu : c'est un **orphelin**, l'autre doit reculer, ce qui peut créer un nouvel orphelin, et ainsi de suite jusqu'au début : l'**effet domino**. Effacer une *réception* est sans danger (le message devient en transit).

Coordonné et CIC

Le checkpointing coordonné évite le domino en forçant une coupe cohérente (deux phases bloquantes, ou piggyback de la demande sans bloquer). Le **CIC** (communication-induced) le fait sans message de contrôle : chaque message porte l'index du checkpoint de l'émetteur ; si l'index reçu dépasse l'index courant, on prend un checkpoint **forcé** avant de traiter le message. Tous les checkpoints de même index forment alors une coupe cohérente.

Message logging pessimiste ou optimiste

On suppose l'exécution « piecewise deterministic » : entre deux événements non déterministes, tout se rejoue pareil. Pessimiste : on journalise chaque réception *avant* de l'exécuter, donc jamais d'orphelin mais un coût synchrone. Optimiste : on journalise par lots, on suit les dépendances, et une panne peut provoquer des retours en cascade.

À noter sur la feuille

Fautes

Du moins grave au plus grave : **crash** (s'arrête, correct avant : fail-stop) < omission (envoi / réception) < timing (hors délai) < response (valeur / transition d'état) < **arbitraire = byzantin**.

k-fault tolerant : survit à k composants en panne (crash : k+1 copies ; byzantin : 2k+1). Stable storage : 1 panne : RAM d'un autre ; transitoires : disque local ; définitives : hors machine (FS répliqué).

Recoverable state : assez d'information pour rejouer jusqu'à une coupe cohérente. **Checkpointing** : (presque) tous reviennent à une coupe cohérente. **Message logging** : seul le fautif revient et rejoue à l'identique avec le journal. En transit : à sauvegarder si canal fiable ; orphelin : réémis donc reçu 2 fois.

Non coordonné, domino, recovery line

Chacun checkpointe quand il veut ; après panne : collecter les dépendances, calculer la **recovery line** ; garder **tous** les checkpoints.

Effet domino : reculer efface un envoi déjà reçu, orphelin, l'autre recule, etc. (effacer une réception = en transit, sans danger).

Graphe de Bhargava-Lian : arête c(i,x) → c(j,y) si m envoyé dans l'intervalle I(i,x) et reçu dans I(j,y), plus arêtes locales. 1) marquer les nœuds volatils des fautifs, 2) atteignabilité, 3) recovery line = dernier nœud non marqué de chaque processus. Slide 22 : ABC impossible car m5 envoyé après A, reçu avant B : domino jusqu'au début.

EN *Rolling back a process may undo a send already received by another, creating an orphan message and forcing a cascading rollback: the domino effect.*

Checkpointing coordonné

Bloquant (2 phases) : l'initiateur checkpoint et envoie request ; chacun : stop, flush, checkpoint provisoire, ack ; tous les acks : commit, checkpoint permanent (garder l'ancien jusqu'au commit). Cohérent, pas de domino, mais latence. Sans blocage : orphelin.

Non bloquant : FIFO, la request arrive avant tout autre message ; ou piggyback de la request sur le premier message. FIFO nécessaire si seul le 1er message porte la request ; inutile si tous portent le numéro (c'est le CIC).

CIC (Briatico)

Index piggybacké ; checkpoint local (index++) ; checkpoint **forcé** : message reçu d'index i > index courant : checkpoint forcé d'index i **avant** de traiter (saut possible 0 → 2). Ligne de recovery k = les checkpoints d'index k (sinon le premier d'index ≥ k) : jamais d'orphelin. i < index : message en transit à sauvegarder (receiver-based).

Exam 2011 : checkpoint local : current := current + 1 ; send (local_state, j, current) à la mémoire stable. N de recovery = **min** sur i du dernier index stocké par i. Index reçu = current - 2 : pas de forcé, mais messages en transit pour N+1..current : pas optimal. Extension : si N < current, pour k = N+1..current : send (M, j, k) à la mémoire stable ; recovery N : renvoyer (local_state, i, N) puis les messages (M, i, N).

CIC contre non coordonné : pas de domino, ligne = même index, 0 message de contrôle.

EN *On receiving a message with a larger index, the process takes a forced checkpoint with that index before delivering it.*

Checkpoint avant / après envoi

Checkpoint **avant** chaque envoi : l'envoi n'est pas dans le checkpoint, orphelins, domino possible. Checkpoint **après** chaque envoi : jamais d'orphelin, pas de domino ; ligne = dernier checkpoint des fautifs + état courant des autres ; GC : garder le dernier seul. Coût : un checkpoint par envoi, plus lourd que le CIC.

2022 : checkpoint à la réception puis renvoi = checkpoint avant envoi : domino ; correctif : renvoyer **puis** checkpointer. Exam 2012 recalculé : avant, ligne P0 655, P1 475, P2 160 (initial), P3 387 (cascade m3, m7, m2, m4, m5) ; après, P1 1046, P3 1026, P0 et P2 à l'état courant.

EN *Checkpointing after each send guarantees that rolling back never undoes a send, so no orphan message can appear.*

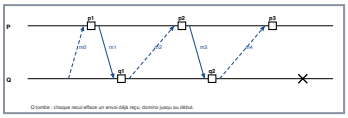
Message logging

PWD (piecewise deterministic) : événements non déterministes (réceptions) identifiables, déterminant stockable, rejeu identique. Processus orphelin : dépend d'un événement ND non journalisé. Condition : pour tout e, ¬Stable(e) ⇒ Depend(e) ⊆ Log(e). Slide 38 : Depend(e) = {P0, P3, P2}, Depend(e') = {P1, P2, P0}. PWD indispensable au logging, inutile au checkpointing.

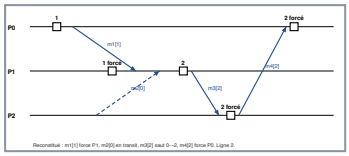
Pessimiste : log AVANT d'exécuter, jamais d'orphelin, seul le fautif repart, simple ; log synchrone coûteux (NVRAM ; si pertes tolérées : log avant l'envoi suivant). **Optimiste** : flush de temps en temps, suivi causal, rollbacks en cascade possibles. Checkpointing : petit ou moyen, surcoût faible ; logging : grande échelle, surcoût moyen à élevé.

EN *Pessimistic logging writes every non-deterministic event to stable storage before executing it, so no orphan process can exist.*

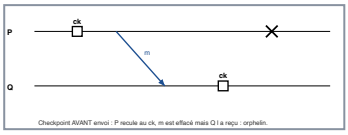
Schémas



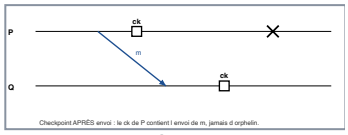
Effet domino : carrés = checkpoints, flèches pointillées = messages Q vers P.



CIC avec index.



Checkpoint avant envoi.



Checkpoint après envoi.

Algorithmes clés

CIC de Briatico (Pi)

```
checkpoint local : index := index + 1 ; save(state, index)
envoi de m : send (m, index)
réception (m, i) :
    si i > index alors index := i ; checkpoint forcé d'index i  // avant de traiter m
    si i < index alors sauvegarder m  // en transit
    traiter m
recovery : N := min_i (dernier index stocké par i) ; tous reviennent au checkpoint d'index N
```

Checkpointing coordonné bloquant

```
initiateur : checkpoint provisoire ; send (request) à tous ; attendre tous les (ack) ; send (commit)
Pi, (request) : stop ; flush ; checkpoint provisoire ; send (ack)
Pi, (commit) : checkpoint permanent ; supprimer l'ancien ; reprendre
```

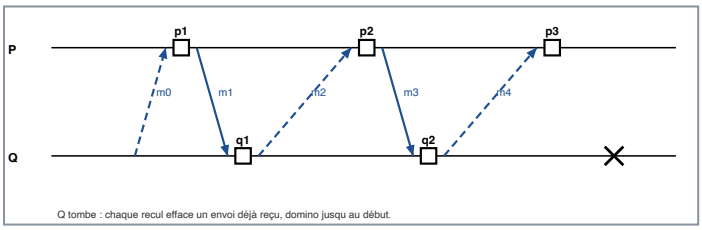
Recovery line (Bhargava-Lian)

```
1. marquer les nœuds volatils des processus fautifs
2. tout nœud atteignable depuis un nœud marqué est marqué
3. recovery line := dernier nœud non marqué de chaque processus
```

Exercices corrigés (chapitre 3)

Effet domino et recovery line (slide 22, graphe de Bhargava-Lian) cours, vérifié sur le site

Énoncé. Trois processus avec checkpoints A, B, C, P2 tombe après C. Peut-on revenir à la coupe {A, B, C} ?



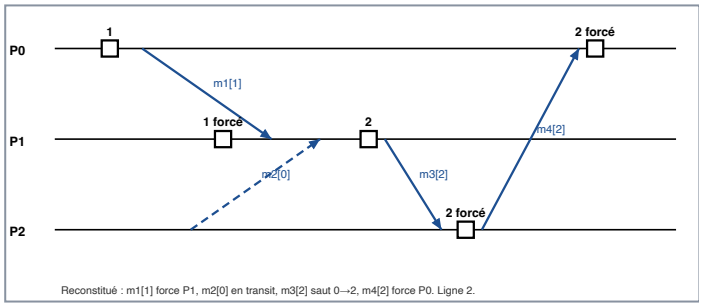
Principe du domino.

1. Construire le graphe : un nœud par checkpoint et un par état volatil, arêtes locales, et une arête c(i,x) → c(j,y) si un message est envoyé dans l'intervalle I(i,x) et reçu dans I(j,y).
2. Marquer l'état volatil du processus fautif, marquer tout ce qui est atteignable. La recovery line est le **dernier nœud non marqué** de chaque processus.
3. Ici m5 est envoyé après A et reçu avant B : la coupe {A, B, C} contient la réception sans l'envoi (orphelin). Reculer résout un orphelin et en crée un autre : **domino jusqu'à l'état initial**.

A ECRIRE La recovery line est l'état initial : domino, car m5 est envoyé après A et reçu avant B.
EN *Rolling back undoes the send of m5, so the receiver must also roll back, up to the initial state.*

CIC : N de recovery et optimalité (exam 2011) annale 2011

Énoncé. Pseudo-code CIC donné : checkpoint local (current := current + 1, send (local_state, j, current) en mémoire stable). Calculer le N de recovery ; que se passe-t-il si l'index reçu vaut current - 2 ?



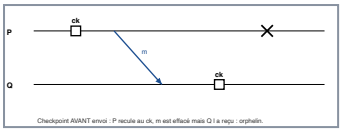
CIC : forcé avant traitement, message en transit si index plus petit.

1. Chaque processus stocke ses checkpoints avec leur index. La recovery line cohérente est l'index **N = min** sur tous les i du dernier index stocké par i.
2. Si le message reçu a l'index **current - 2** : pas de checkpoint forcé (index plus petit), mais le message est en transit pour tous les index N+1..current : s'il n'est pas sauvegardé il est perdu au recovery. L'algorithme n'est donc **pas optimal**.
3. **Extension** : si N < current, pour k = N+1..current : send (M, j, k) en mémoire stable. Au recovery d'index N : renvoyer (local_state, i, N) puis les messages (M, i, N).

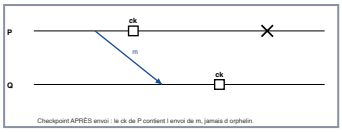
A ECRIRE N = min des derniers index stockés ; il faut aussi stocker les messages en transit (receiver-based).
EN *The recovery index is the minimum of the last stored indices; in-transit messages must be logged too.*

Checkpoint avant ou après chaque envoi (exam 2012, 2022) annales 2012 et 2022, valeurs recalculées

Énoncé. Quatre processus, messages m1 à m10, checkpoint avant chaque envoi, puis après chaque envoi : graphe de dépendances, recovery line, domino ? 2022 : P1 et P2 se renvoient m, checkpoint à la réception avant de renvoyer.



Avant : orphelin.



Après : cohérent.

1. **Avant chaque envoi** : l'envoi n'est pas dans le checkpoint qui le précède, donc revenir à ce checkpoint efface l'envoi, alors que le receveur peut avoir checkpointé après la réception : orphelin, cascade. Sur l'exam 2012 recalculé : ligne P0 655, P1 475, P2 160 (état initial), P3 387, cascade m3, m7, m2, m4, m5.
2. **Après chaque envoi** : le checkpoint contient l'envoi, donc jamais d'orphelin et jamais de domino. Ligne : dernier checkpoint des fautifs et état courant des autres (P1 1046, P3 1026, P0 et P2 à l'état courant). Garbage collection : garder seulement le dernier checkpoint.
3. Inconvénient : un checkpoint par envoi, très coûteux ; le CIC ne force que quand c'est nécessaire.
4. **2022** : checkpoint à la réception puis renvoi = checkpoint avant envoi : coupe des deux derniers checkpoints incohérente, domino jusqu'au début. Correctif : **renvoyer m puis checkpointer**.

A ECRIRE Checkpoint après chaque envoi : pas d'orphelin, pas de domino, mais un checkpoint par envoi.
EN *Checkpointing after each send guarantees that rolling back never undoes a send.*

Message logging : orphelins et slide 38 cours

Énoncé. Condition de non orphelin et exemple du slide 38.

1. Un processus est **orphelin** s'il dépend d'un événement non déterministe non journalisé. Condition : pour tout événement e, si e n'est pas stable alors Depend(e) ⊆ Log(e).
2. Slide 38 : Depend(e) = {P0, P3, P2} et Depend(e') = {P1, P2, P0}.
3. **Pessimiste** : log avant exécution, jamais d'orphelin, seul le fautif repart. **Optimiste** : flush périodique, rollbacks en cascade possibles.

A ECRIRE Pessimiste : log avant exécution, un seul processus recule ; optimiste : suivi causal.
EN *Pessimistic logging records each event before it executes, so no process can become an orphan.*

Réponses types des annales

Q. CIC, exam 2011 : N de recovery et optimalité

R. N = min des derniers index stockés. Si l'index reçu vaut current - 2, aucun checkpoint forcé mais des messages sont en transit pour N+1..current : pas optimal. Extension : stocker (M, j, k) pour k = N+1..current et les renvoyer au recovery.

EN *The recovery index is the minimum over processes of their last stored index; in-transit messages must also be logged.*

Q. Checkpoint avant contre après chaque envoi

R. Avant : l'envoi n'est pas inclus, orphelins et domino possibles. Après : jamais d'orphelin ni de domino, mais un checkpoint par envoi. Le CIC fait mieux avec peu de checkpoints forcés.

EN *Checkpointing after each send never creates orphans, but it costs one checkpoint per send.*

Q. Exam 2022 : checkpoint à la réception puis renvoi de m

R. Le renvoi est après le checkpoint donc l'envoi n'y figure pas : orphelin, dépendances en chaîne, domino jusqu'au début. Correctif : renvoyer m puis checkpointer.

EN *Send first, then take the checkpoint, so the checkpoint includes the send.*

Q. Pourquoi le message logging pessimiste évite-t-il les orphelins ?

R. Chaque réception est journalisée avant exécution : aucun événement non déterministe non journalisé ne peut influencer un autre processus. Seul le fautif revient et rejoue.

EN *Every non-deterministic event is logged before execution, so no process can depend on a lost event.*

PIEGES **à éviter**

- Effacer une réception donne un message en transit (sans danger), effacer un envoi déjà reçu donne un orphelin.
- CIC : le checkpoint forcé se prend AVANT de traiter le message.
- Recovery line = checkpoints d'index k, ou le premier d'index ≥ k si un processus n'en a pas.
- Coordonné sans blocage : orphelin possible.
- Messages en transit à sauvegarder si le canal est fiable ; orphelins réémis donc reçus deux fois.
- PWD est indispensable au logging, inutile au checkpointing.

Chapitre 4 | Communications de groupe

TOMBE Deux propriétés d'ordre causal safety / liveness (2011, 2012-13), total order par séquenceur sans FIFO et panne du séquenceur (2012-13, 2013-14), CBCAST à vecteurs (2017-18, 2019), liens perfect à partir de fair-loss (2019).

Explications

Une pile de garanties de diffusion

On part de liens perdants (fair-loss), on les rend têtus (stubborn), puis parfaits (perfect, sans perte ni duplication). Au-dessus : **BEB** (le meilleur effort : un correct envoie à tous les corrects), **RB** (tout ou rien parmi les corrects, grâce à un détecteur P qui déclenche le relais), **URB** (tout ou rien même pour les processus fautifs qui ont livré). Chaque étage se construit sur le précédent.

Les ordres de livraison

FIFO : mes messages sont livrés dans l'ordre où je les ai envoyés. **Causal** : si m1 a causé m2, tous livrent m1 avant m2 (se réalise avec des vecteurs : CBCAST). **Total** : tous livrent dans le même ordre, mais pas forcément causal (se réalise avec un séquenceur ou des horloges de Lamport).

Safety et liveness : l'exemple du chat

m1 = une question, m2 = la réponse. Règle (a) : si on livre m1 et m2, m1 d'abord (safety, rien de mauvais n'arrive). Règle (b) : livrer m2 impose que m1 soit déjà livré (plus forte, c'est la spécification du broadcast causal). Livrer la réponse sans la question respecte (a) mais viole (b).

À noter sur la feuille

Safety et liveness

Safety = rien de mauvais n'arrive (violation visible à un instant t). **Liveness** = quelque chose de bon finit par arriver.

Causal (exam) : m1 = question, m2 = réponse (m1 → m2). (a) si on livre m1 ET m2, m1 d'abord : safety. (b) si on livre m2, m1 est déjà livré : plus forte, c'est la spec CO (CB slide 49). Livrer la réponse seule : (a) respectée, (b) violée.

m1 → m2 vient de : C1 FIFO, C2 réseau (livre m1 puis diffuse m2), C3 transitivité.

EN *Safety: nothing bad ever happens. Liveness: something good eventually happens.*

BEB, RB, URB

BEB : bebBroadcast = pp2pSend à tous. BEB1 (pi ET pj corrects : livré) liveness, BEB2 pas de duplication (safety), BEB3 pas de création (safety).

RB = BEB + tout ou rien. Détecteur P : PFD1 strong completeness (liveness), PFD2 strong accuracy (safety). RB4 agreement : livré par un **correct** ⇒ livré par tous les corrects. Lazy (slide 68) : from[pi] ; crash(pi) : rediffuser from[pi] ; pi déjà suspecté : relais immédiat. EXO 2 : P2 relaie sur crash P0, P3 sur crash P1, P4 livre ; P2, P3, P4 livrent une seule fois.

URB : si un processus (même fautif) livre m ⇒ tous les corrects livrent. All-Ack : relais immédiat, livrer quand correct ⊆ ack_m (N^2 messages). Majority-Ack sans détecteur si f < N/2.

EN *Reliable broadcast: if a correct process delivers m, then every correct process delivers m.*

CBCAST (vecteurs de diffusion)

VC[i] = messages diffusés par pi, VC[j] = messages de pj livrés. Envoyer VC puis VC[self]++. Livrer si **VC_m ≤ VC** puis VC[émetteur]++ (**pas de max**). VC ne change qu'à la livraison.

EXO 3 : m1 000, m2 100, m3 110, m5 110, m4 210 ; P2 livre m1 m2 m5 m3 m4 ; crash de P1 pendant m3 : sans RB, m4 bloqué sur P2 ; avec RB, P3 relaie m1 et m3.

EN *A message is delivered once its vector is less than or equal to the local one; the vector is updated at delivery only.*

Total order

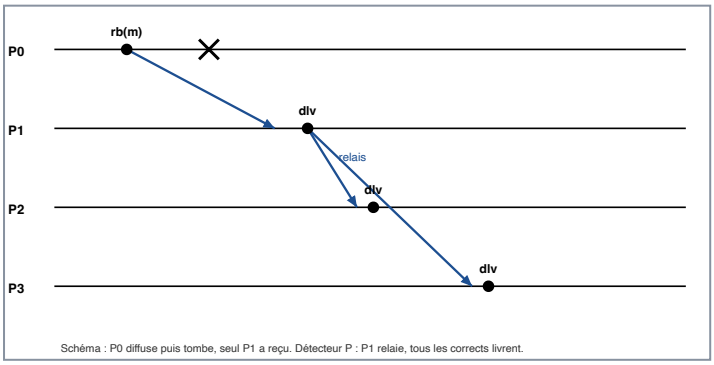
Même ordre partout, pas forcément l'ordre d'envoi. **Séquenceur** : r := 0 ; order(i, S) : attendre m ET S = r ; S numérote à l'arrivée. Q1 sans FIFO : m2 double m1, tous livrent m2 puis m1 (pas causal). Q2 FIFO ⇒ causal. Q4 : 2N messages, 2 étapes. Q5 : m envoyé seulement à S : N+1 messages, FIFO pi → S seulement.

Panne de S : élection (détecteur + élection de leader), collecte des orders reçus, reprise à max + 1, orders diffusés en URB.

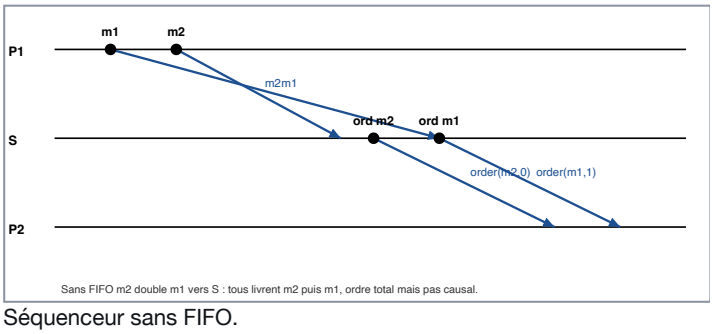
Mysterious broadcast (EXO 4) : Lamport + FIFO, livrer ts ≤ min_q LC[q] : ordre total causal ; liveness : chacun diffuse régulièrement ; sans FIFO : plus aucun ordre.

EN *The sequencer assigns a sequence number on arrival; receivers deliver in sequence-number order.*

Schémas



Reliable broadcast avec relais.



Séquenceur sans FIFO.

Algorithmes clés

Reliable Broadcast lazy (slide 68)

```
Init: delivered := ∅ ; correct := Π ; from[pi] := ∅ pour tout pi
rbBroadcast(m): bebBroadcast((Data, self, m))
bebDeliver(pj, (Data, s, m)): si m ∉ delivered :
delivered ∪= {m} ; rbDeliver(s, m) ; from[s] ∪= {m}
                                    si s ∉ correct :
bebBroadcast((Data, s, m))
crash(pi): correct \= {pi} ; pour tout m ∈ from[pi] :
bebBroadcast((Data, pi, m))
```

CBCAST (pi)

```
cbcast(m): envoyer ⟨m, VC⟩ à tous ; VC[i] := VC[i] + 1
réception ⟨m, VCm⟩ de pj: attendre  VCm ≤ VC
(composante par composante)
        livrer m ; VC[j] := VC[j] + 1  // pas de max
```

Total order par séquenceur

```
to-broadcast(m): envoyer m à tous et à S
S, réception de m: envoyer ⟨order, id(m), seq⟩ à tous ;
seq := seq + 1
pi: r := 0 ; livrer m dès que m reçu et ⟨order, id(m), r⟩ reçu ; r := r + 1
```

Exercices corrigés (chapitre 4)

Deux propriétés causales (annales 2011 et 2012-13)

annales

Énoncé. m1 est une question, m2 la réponse. Les propriétés (a) et (b) sont-elles équivalentes ? Safety ou liveness ?

1. (a) : si un processus livre m1 et m2, il livre m1 d'abord. C'est une **safety** : une violation (m2 puis m1) est visible à un instant précis.
2. (b) : si un processus livre m2, alors m1 a déjà été livré. Plus forte : elle interdit aussi de livrer la réponse seule. C'est la spécification du causal broadcast (CB, slide 49).
3. Livrer la réponse seule : (a) respectée (m1 jamais livré), (b) violée. Donc (b) implique (a), pas l'inverse.

A ECRIRE (b) implique (a) ; elles diffèrent quand m1 n'est jamais livré. Les deux sont des safety.
EN *Property (b) is stronger: it forbids delivering m2 unless m1 has already been delivered.*

EXO 2 (slide 69) : reliable broadcast avec détecteur

corrigé officiel concordant

Énoncé. Cinq processus, P0 puis P1 tombent pendant leur diffusion. Qui relaie, qui livre, combien de fois ?

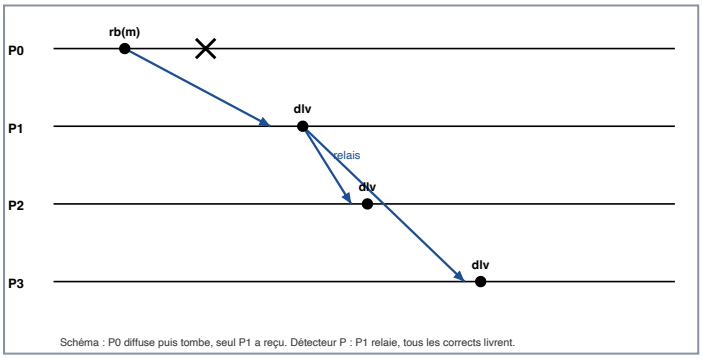


Schéma de principe du relais.

1. RB lazy : chacun garde from[pi] ; sur crash(pi) il rediffuse tout from[pi] ; si l'émetteur est déjà suspecté, il relaie immédiatement.
2. Résultat : P2 relaie sur le crash de P0, P3 relaie sur le crash de P1, P4 livre. P2, P3 et P4 livrent chaque message **une seule fois** (ensemble delivered).
3. RB4 agreement : livré par un correct implique livré par tous les corrects.

A ECRIRE P2 relaie sur crash(P0), P3 sur crash(P1), P4 livre ; P2, P3, P4 livrent une fois.
EN *When the sender is suspected, a correct process that delivered the message relays it to everybody.*

EXO 3 (slide 78) : CBCAST et crash de P1

corrigé officiel partie causale

Énoncé. Messages m1 (000), m2 (100), m3 (110), m5 (110), m4 (210) ; P2 livre m1 m2 m5 m3 m4. Que se passe-t-il si P1 tombe pendant m3 ?

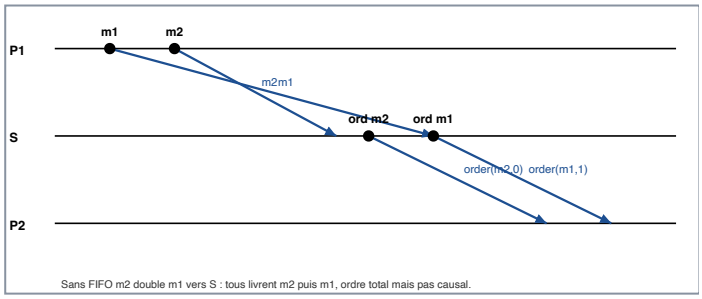
1. Le vecteur porté par un message est le VC de l'émetteur à l'envoi. Il est livrable chez pj quand VC_m ≤ VC(pj) composante par composante ; la livraison incrémente VC[émetteur] (jamais de max).
2. Ordre de livraison chez P2 : m1, m2, m5, m3, m4, chacun attendant les messages dont il dépend.
3. **Crash de P1 pendant m3, sans RB** : P2 n'a pas reçu m3 donc m4 (210) reste bloqué. **Avec RB** : P3 relaie m1 et m3, P2 les reçoit et débloque m4.

A ECRIRE CBCAST seul peut bloquer sur un message perdu ; un reliable broadcast dessous garantit l'arrivée.
EN *Without reliable broadcast a lost message blocks every message that causally depends on it.*

Total order par séquenceur (annales 2012-13 Q1 à Q5, 2013-14)

annales

Énoncé. Séquenceur S numérote les messages ; sans FIFO, avec FIFO ; coût ; panne de S.



m2 double m1 vers S.

1. Principe : r := 0 chez chaque processus ; livrer m quand m est reçu et que order(i, S) avec S = r est reçu ; S numérote à l'arrivée.
2. **Q1, sans FIFO** : m2 double m1 vers S, S numérote m2 puis m1, tous livrent m2 puis m1 : total mais pas causal.
3. **Q2** : FIFO entre l'émetteur et S implique causal. **Q4** : 2N messages en 2 étapes (N vers S et N d'ordres). **Q5** : si m n'est envoyé qu'à S, N+1 messages ; FIFO seulement pi vers S.
4. **Panne de S** : détecter, élire un nouveau séquenceur, collecter les ordres déjà reçus, reprendre à max + 1, diffuser les ordres en URB.

A ECRIRE FIFO source-séquenceur donne le causal ; coût 2N messages ; reprise à max + 1 après élection.
EN *The sequencer assigns increasing numbers on arrival; all processes deliver in that numeric order.*

EXO 4 : mysterious broadcast cours

Énoncé. Livrer quand ts ≤ min_q LC[q] : quel ordre ? Que faut-il supposer ?

1. Chaque processus diffuse avec son horloge de Lamport et garde LC[q], la dernière estampille vue de q. Il livre les messages dont l'estampille est ≤ le minimum des LC[q] : plus aucun message plus petit ne peut arriver (FIFO).
2. Ordre obtenu : **total et causal** (estampille puis id).
3. Vivacité : il faut que chacun diffuse régulièrement (sinon min bloqué). Sans FIFO : plus aucun ordre garanti.

A ECRIRE Ordre total causal par estampille, sous FIFO et diffusions régulières.
EN *Messages are delivered in timestamp order once no smaller timestamp can still arrive.*

Réponses types des annales

Q. Les deux propriétés causales sont-elles équivalentes ?

R. (a) : si on livre m1 et m2, m1 d'abord (safety). (b) : si on livre m2, alors m1 déjà livré (plus forte, c'est CO). (b) implique (a) ; (a) n'interdit pas de livrer la réponse seule.

EN *The second property is stronger: it forbids delivering the answer without the question.*

Q. Séquenceur sans FIFO : contre-exemple causal

R. m2 envoyé après m1 par la même source mais m2 double m1 vers S : S numérote m2 puis m1, tous livrent m2 m1. L'ordre total est respecté mais pas l'ordre causal. Avec FIFO source → S, l'ordre total respecte la causalité.

EN *If m2 overtakes m1 on the way to the sequencer, everyone delivers m2 then m1: total order without causal order.*

Q. Panne du séquenceur : que faire ?

R. Détecter (détecteur de pannes), élire un nouveau séquenceur (leader election), collecter les order déjà reçus, reprendre la numérotation à max + 1, diffuser les order en URB pour ne pas en perdre.

EN *Elect a new sequencer, collect the orders already received, resume at the maximum plus one, and broadcast orders uniformly.*

Q. CBCAST : quand livrer ?

R. Quand VC_m ≤ VC : toutes les diffusions dont m dépend ont déjà été livrées localement, et c'est le suivant attendu de l'émetteur. Puis VC[émetteur]++ (pas de max).

EN *Deliver once every message that causally precedes m has already been delivered locally.*

PIEGES **à éviter**

- URB engage les processus fautifs aussi, RB seulement les corrects.
- CBCAST : VC[émetteur]++ à la livraison, pas un max ; le vecteur ne change pas à la réception.
- Total order ne dit rien sur l'ordre d'envoi ni la causalité.
- Le séquenceur sans FIFO ne donne pas l'ordre causal.
- RB lazy : relayer seulement quand l'émetteur est suspecté (détecteur P).
- Ne pas confondre l'état d'attente de livraison (hold) et la réception.

Chapitre 5 | Détecteurs de fautes et consensus

TOMBE Consensus uniforme contre régulier (2011, 2012, 2019), flooding en N-1 étapes, détecteurs ◇P et élection de leader (2011, 2012), commit de transaction (2019).

Explications

Le consensus et pourquoi il est difficile

Chaque processus propose une valeur, tous doivent décider la même, parmi celles proposées, exactement une fois, et tout processus correct doit finir par décider. FLP : en asynchrone pur, même avec un seul crash possible, aucun algorithme déterministe ne garantit à la fois l'accord et la terminaison, car on ne peut pas distinguer un processus lent d'un processus mort.

Détecteurs de fautes

Un détecteur dit à chaque processus qui il soupçonne. **Completeness** : les morts sont finalement soupçonnés. **Accuracy** : les vivants ne le sont pas (forte : jamais ; finale : à partir d'un moment). P : synchrone, parfait. ◇P : partiellement synchrone, il se trompe un temps puis se stabilise, en augmentant son délai à chaque erreur. ◇S est le plus faible qui permet le consensus.

Consensus hiérarchique et uniforme

Hiérarchique : les processus sont classés par rang ; à son tour, le chef décide sa valeur et la diffuse, les autres l'adoptent ou suspectent le chef. Il est correct mais **non uniforme** : le chef décide avant que personne n'ait reçu, donc il peut décider puis crasher sans que les autres le suivent. Le **consensus uniforme** exige l'accord de tous, y compris les fautifs ; il faut des acquittements avant de décider.

Paxos, CAP

Paxos suppose seulement qu'un leader existe à terme (Ω) ; avec un quorum de majorité il reste sûr même si plusieurs processus se croient leaders. CAP : si le réseau se partitionne, on choisit la cohérence ou la disponibilité, pas les deux. La cohérence à terme (eventual consistency) n'est pas la cohérence de CAP.

À noter sur la feuille

Consensus et FLP

4 propriétés : **terminaison** (liveness : tout correct décide), **accord**, **validité** (la valeur décidée a été proposée), **intégrité** (une seule décision). Uniforme : l'accord vaut pour **tous** les processus, y compris ceux qui crashent après avoir décidé.

FLP : consensus impossible en asynchrone pur, même avec 1 crash (mort ≠ lent). Sorties : détecteur de pannes, borner f (quorum f+1 parmi 2f+1), probabiliste.

EN *FLP: no deterministic algorithm solves consensus in an asynchronous system if even one process may crash.*

Détecteurs de fautes

Completeness (les morts sont suspectés) forte / faible ; **accuracy** (les vivants ne sont pas suspectés) forte / faible ; ◇ = à terme. On ne garde que la completeness forte (boosting).

P : synchrone, délai max δ connu ; strong completeness (liveness) + strong accuracy (safety). ◇**P** : partiellement synchrone ; sur restore, T augmente (T += Δ) ; strong completeness + eventual strong accuracy. Impossible en asynchrone pur ; « pas plus d'un jour » = synchrone formellement, inutile en pratique.

Classes P, S, ◇P, ◇S ; ◇**S** est le plus faible qui résout le consensus. Leader election : LE2 (leader ⇒ anciens morts) est une safety, besoin de P ; **Ω** : à terme le même leader, faisable avec ◇P.

EN *An eventually perfect failure detector may suspect correct processes for a while, but eventually every crashed process is suspected and no correct one is.*

Consensus hiérarchique (algorithme I et 5.2)

Tour i : chef p_i décide sa proposition courante et la diffuse (beb) ; les autres l'adoptent ou suspectent le chef. Au plus n tours. Algorithme 5.2 (slides 43-44) : **if (r < rank(self)) and (r > proposer) then proposal := v ; proposer := r ; delivered[r] := true** (toujours). Première moitié : ne rien adopter après son propre tour. Seconde : un ancien message ne doit pas écraser un plus récent (message en retard).

Correct mais non uniforme : le chef décide avant de diffuser, il peut décider puis crasher sans que personne d'autre ne le suive. Le scénario du message en retard donne un vrai désaccord seulement avec n = 4 (le message doit arriver entre deux tours).

EN *The hierarchical algorithm is correct but not uniform: the leader may decide and crash before anyone else adopts its value.*

Consensus uniforme (algorithme II)

Messages PROPOSAL et ACK. Ack si rank(p) ≥ round. Adoption de proposed[round] seulement à la **détection du crash** du chef. Décision par rbBroadcast(DECIDED) quand **detectedranks** ∪ **ackranks = {1..N}**. Cas « décision b » : le chef 1 crashe en pleine diffusion, p3 a reçu a, p2 non, le tour 2 décide b.

Annales : « le flooding uniforme en N-1 étapes est-il correct ? » (2011), différence flooding / hiérarchique avec deux diagrammes et les complexités (2012). Dessiner les deux exécutions et compter messages et étapes.

EN *Uniform consensus: no process, even one that crashes later, may decide differently from the others.*

Chandra-Toueg ◇S, Paxos, CAP

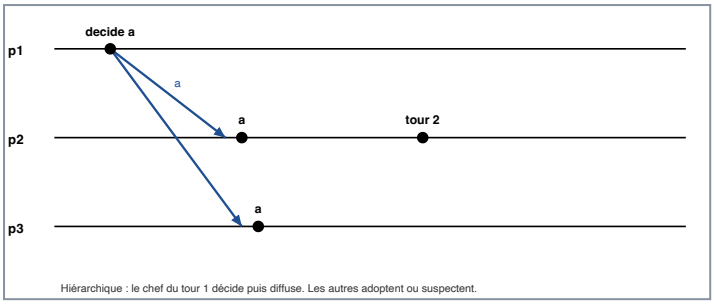
Chandra-Toueg ◇**S** : tours à coordinateur tournant, 4 phases : estimation + timestamp, proposition du plus grand timestamp parmi une majorité, ack ou nack, décision par RB si majorité d'ack sans nack. **f < n/2** requis même avec ◇P (argument de partition). **S** : 3 phases, n-1 tours, tolère n-1 pannes.

Paxos : suppose seulement un leader à terme (Ω), plus complexe car pas de vue fiable ; à chaque tour, le rang le plus haut gagne (alors que le plus bas gagne en hiérarchique : même principe, échelle de temps différente). Ring Paxos : coordinateur dédié.

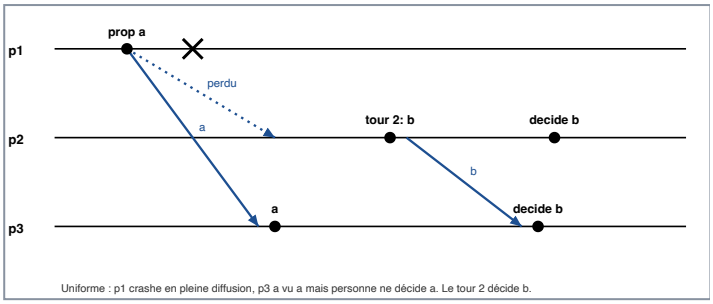
CAP : en cas de partition, choisir C ou A. CP = Paxos ; CA = pas de partition. Cohérence à terme ≠ C de CAP. Horloges vectorielles pour détecter la concurrence et fusionner (siblings). Dynamo (papier SOSP 2007) expose les vecteurs ; DynamoDB (service AWS) fait last-writer-wins.

EN *During a network partition, a system must choose between consistency and availability.*

Schémas



Hiérarchique, premier tour.



Uniforme, décision b.

Algorithmes clés

Consensus hiérarchique 5.2 (règle clé)

```
// tour r = 1..N, le chef du tour r est le processus de rang r
tour r : si r = rank(self) et proposal ≠ ⊥ :
decide(proposal) ; bebBroadcast((DECIDED, proposal))
bebDeliver(pr, (DECIDED, v)):
    si (r < rank(self)) et (r > proposer) : proposal := v ; proposer := r
    delivered[r] := vrai  // toujours
```

◇P à timeout croissant (algo 2.6)

```
Timeout: si (alive ∩ suspected) ≠ ∅ : period := period + Δ
    pour tout pi : si pi ∉ alive et pi ∉ suspected :
suspected ∪= {pi} ; suspect(pi)
                sinon si pi ∈ alive et pi ∈ suspected :
suspected \= {pi} ; restore(pi)
                send(pi, HEARTBEAT)
    alive := ∅ ; startTimer(period)
Deliver HEARTBEAT de q: alive ∪= {q}
```

Chandra-Toueg ◇S (tour r, coordinateur c)

```
1. chacun envoie (estimation, timestamp) à c
2. c attend une majorité, prend l'estimation de plus grand timestamp, la diffuse
3. chacun répond ack (si reçu) ou nack (si c suspecté)
4. c reçoit une majorité d'ack sans nack : RB(decide)
// f < n/2
```

Exercices corrigés (chapitre 5)

Slides 20-21 : ◇P avec un seul compteur `rédigé, voir le site`

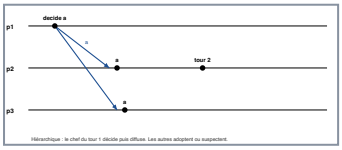
Énoncé. Un seul compteur period sert aux heartbeats et à la détection. Est-ce un bon algorithme ? Sans panne mais Δ mal calibré ? Délai entre deux heartbeats, délai maximal de détection ?

1. Correct pour ◇P mais couplé : quand period augmente, on envoie aussi moins de heartbeats.
2. Sans panne et TimeDelay trop petit : chaque processus suspecte à tort les autres au premier timeout, les restaure au suivant et augmente la période de Δ ; après un nombre fini d'erreurs la période dépasse le délai réel et les erreurs cessent.
3. Délai entre deux heartbeats : TimeDelay + j·Δ où j est le nombre de timeouts avec un suspect trouvé vivant. Détection au plus δ + 2·period avec period ≤ max(TimeDelay, δ + Δ).

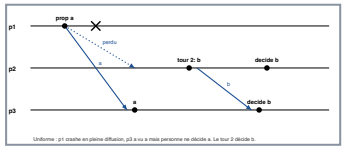
`A ECRIRE` Le timeout croît à chaque restore, donc finit par dépasser le délai réel : eventual strong accuracy.
`EN` *The timeout grows after every false suspicion, so eventually it exceeds the real delay.*

Uniforme contre régulier (annales 2011, 2012, 2019) `annales, scénarios du site`

Énoncé. Différence entre consensus uniforme et régulier, scénario où le hiérarchique viole l'uniformité, deux diagrammes.



Hiérarchique : le chef décide puis diffuse.



Uniforme : décision b.

1. **Régulier** : seuls les processus corrects s'accordent. **Uniforme** : tout processus qui décide, même s'il crashe ensuite, décide comme les autres.
2. Hiérarchique : p1 décide a puis crashe avant d'avoir envoyé a à personne ; p2 et p3 décident autre chose : accord entre les corrects, mais p1 a décidé différemment : non uniforme.
3. Algorithme II (uniforme) : le chef envoie PROPOSAL, chacun répond ACK ; la décision se prend quand detectedranks ∪ ackranks = {1..N}, annoncée par RB. Cas « décision b » : le chef 1 crashe en pleine diffusion, p3 a reçu a, p2 non, personne ne décide a, le tour 2 décide b.
4. Message en retard (algorithme 5.2) : tester (r < rank) et (r > proposer) ; un vrai désaccord n'apparaît qu'avec n = 4.

`A ECRIRE` Uniforme : aucun processus, même fautif, ne décide autre chose ; le hiérarchique ne l'assure pas.
`EN` *In uniform consensus even a process that crashes later cannot have decided differently.*

CAP, quorums et Dynamo (Exo 5 et homework) `cours`

Énoncé. Partition réseau : que choisir ? Cohérence à terme ?

1. En cas de partition, on choisit C (refuser des requêtes) ou A (répondre, éventuellement avec des données périmées). CP = Paxos ; CA = pas de partition.
2. Cohérence à terme : sans nouvelle écriture, les copies convergent ; ce n'est pas la cohérence de CAP. Quorums N, R, W : R + W > N garantit qu'une lecture voit la dernière écriture.
3. Horloges vectorielles pour détecter les écritures concurrentes (siblings). Dynamo (papier SOSP 2007) les expose ; DynamoDB (service AWS) fait last-writer-wins.

`A ECRIRE` Partition : cohérence ou disponibilité ; R + W > N pour des lectures fraîches.
`EN` *During a partition the system must choose between consistency and availability.*

Réponses types des annales

Q. Consensus uniforme contre régulier : idée clé

R. Régulier : seuls les processus corrects s'accordent. Uniforme : aucun processus, même un qui crashe ensuite, ne décide différemment. Le hiérarchique décide avant de diffuser donc n'est pas uniforme ; l'algorithme II attend les acks de tous les non détectés avant de décider.

EN *Regular consensus constrains correct processes only; uniform consensus constrains every process that decides, even faulty ones.*

Q. ◇P en asynchrone pur ? « pas plus d'un jour » ?

R. Impossible en asynchrone pur (FLP). « Au plus un jour » est formellement une borne donc le système est synchrone, mais une borne trop grande rend le détecteur inutile en pratique.

EN *An eventually perfect detector needs partial synchrony; a one-day bound makes the system synchronous in theory but useless in practice.*

Q. Pourquoi f < n/2 pour ◇S ou ◇P ?

R. Avec ◇ le détecteur peut se tromper : si f ≥ n/2, une partition peut produire deux groupes qui décident chacun une valeur différente. Il faut une majorité qui s'intersecte.

EN *With an unreliable detector, a partition into two halves could decide two different values, so a majority is required.*

Q. Commit de transaction et consensus

R. Atomic commit = consensus où un Non impose abort. 2PC est bloquant, 3PC ne l'est pas sous ses hypothèses ; les systèmes réels répliquent chaque valeur par un consensus (Paxos).

EN *Atomic commit is a consensus problem in which a single No vote forces abort.*

PIEGES **à éviter**

- Uniforme concerne les processus qui décident puis crashent.
- Le consensus hiérarchique est non uniforme : le chef décide avant de diffuser.
- ◇P : le délai augmente à chaque restore, jamais une borne fixe.
- Message en retard dans le consensus hiérarchique : tester (r < rank) et (r > proposer).
- CAP : cohérence à terme n'est pas la cohérence de CAP.
- DynamoDB (service) fait last-writer-wins, pas de vecteurs ; c'est Dynamo (papier) qui les expose.

Chapitre 6 | Transactions distribuées et état global

TOMBE Terminaison (Misra : combien de tours, 2017 ; jeton sur anneau pondéré, 2019 ; quand détecter, 2012-13), commit de transaction (2019). Le dernier cours est aussi couvert par le TD fait en cours (exos 1 à 3).

Explications

Deux sujets en un chapitre

Première partie : une transaction touche plusieurs sites et doit être atomique. Il faut un protocole où tous valident ou tous annulent (2PC, 3PC) et, séparément, un contrôle de concurrence pour l'isolation. Seconde partie : détecter un état global (fin de l'application, interblocage) sans voir tout le système, par vagues, par jeton ou par sondes.

Pourquoi une vague de comptage ne suffit pas

Une vague lit les compteurs de chaque processus à un instant différent. Si P0 est lu avant d'envoyer un message et P2 après l'avoir reçu, la vague voit une réception sans envoi ; un autre message en transit compense, et on trouve S = R alors qu'un message circule encore. Mattern évite le problème avec deux vagues qui doivent donner les mêmes totaux, la seconde commençant après la fin de la première.

Misra et Dijkstra-Feijen-van Gasteren : même idée, deux règles

Un jeton fait le tour de l'anneau et collecte l'information « je suis resté passif depuis ta dernière visite ». Dans Misra, toute réception noircit, le jeton compte les processus blancs consécutifs et il faut connaître n. Dans DFvG, seul un envoi vers un numéro plus élevé (que le jeton a déjà visité) noircit ; le jeton et les processus se colorent, et l'initiateur n'a besoin que de la couleur.

À noter sur la feuille

ACID et commit

Atomicity (tout ou rien), **C**onsistency (règles d'intégrité), **I**solation (pas d'effet de bord entre transactions concurrentes : sérialisabilité), **D**urability. Commit protocol = atomicité répartie ; contrôle de concurrence (pessimiste : verrous, cycle puis avorter ; optimiste : graphe de précédence examiné à la fin ; MVCC) = isolation. Lecture sale : si T1 avorte, T2 doit avorter.

Propriétés d'un protocole de commit : **Agreement** (un seul Non impose abort), **Termination** (tout correct décide), **Validity** (tous Oui sans panne : commit). Base répliquée : chaque valeur du 2PC (PREPARE:OUI, COMMIT) est décidée par MultiPaxos entre les répliques d'un shard.

EN *A commit protocol is a consensus problem in which a single No vote forces abort.*

2PC et 3PC

2PC : phase 1 : le coordinateur écrit Preparation, envoie Prepare, les participants écrivent Ready dans le log (mémoire stable) et votent Oui ou Non. Phase 2 : Commit si tous Oui, sinon Abort, puis Done. **Bloquant** : si le coordinateur tombe entre les phases, les participants prêts sont **indécis**, ils attendent son retour.

3PC (Skeen) : phase Prepare / Ack et état tampon **p**. C(s) = ensemble de concurrence : blocage si C(s) contient commit et abort. Timeouts : cohort en **w** : abort ; en **p** : commit. Coordinateur au retour : en w1 abort, en p1 commit. Hypothèses : **au plus un site en panne**, timeouts exacts, pas de partition. 3 aller-retours contre 2.

EN *2PC blocks if the coordinator fails between the two phases; 3PC adds a prepared state so that survivors can decide on their own.*

Terminaison : 4 compteurs de Mattern

Vague : lit les compteurs s (envoyés) et r (reçus) de chaque processus. Une vague S = R ne suffit pas (coupe incohérente). **Deux vagues, la seconde après la fin de la première** : si **E1 = R1 = E2 = R2** alors terminé. Au plus 2 vagues après la terminaison.

Preuve : $d1 < f1 < d2 < f2$; $R(t) \le E(t)$; $R1 \le R(f1)$; $E(d2) \le E2$; $R1 = E2$ donne $E(f1) \le E(d2) \le E2 = R1 \le R(f1) \le E(f1)$, donc **E(f1) = R(f1)**.

EN *If two consecutive waves report E1 = R1 = E2 = R2, no message was in transit at the end of the first wave.*

Terminaison : Misra et Dijkstra-Feijen-van Gasteren

Misra : anneau unidirectionnel, canaux FIFO, messages et jeton sur l'anneau. Tous blancs ; réception d'un message : noir. Pi : si noir token := 1 sinon token + 1 ; devient blanc ; transmet. P0 : blanc et token = n : **terminé**, sinon relance token = 1. Au plus **2 tours** après la terminaison. Traiter le jeton seulement passif. Sans FIFO : faux.

DFvG : anneau $0 \to n-1 \to \dots \to 1 \to 0$, communications synchrones (ou bornées par D : attendre D avant le jeton). R1 : un envoi vers un numéro plus élevé noircit l'émetteur. R2 : un noir qui envoie le jeton le rend noir. R3 : il redevient blanc. Initiateur : jeton blanc et lui blanc : terminé. Un tour suffit si personne n'a noirci. Sans n ni anneau : **Dijkstra-Scholten** (acks, parent, déficit).

EN *Misra's token counts consecutive white processes; it reaches n only if no message was received during a full round.*

Interblocage

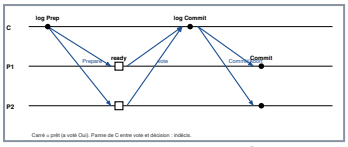
WFG (wait-for graph). **ET** (ressources) : cycle orienté. **OU** (communication) : un seul message débloque.

Chandy-Misra-Haas ET : sonde P(i, s, r). Reçue en k : si w[k], $k \neq i$, ¬depend[k, i] : envoyer P(i, k, j) aux successeurs, depend[k, i] := vrai ; si k = i : interbloqué. L'initiateur doit être sur un circuit.

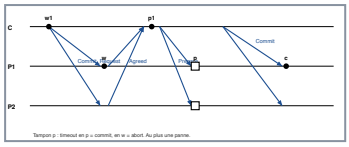
Chandy-Misra-Haas OU (probe-echo) : un non-attendant ignore ; 1ère sonde : parent + transmission ; suivantes : ack ; tous les acks : ack au parent ; initiateur avec tous les acks : **interbloqué**. Un cycle ne suffit pas pour le OU (un successeur libre débloque tout).

EN *In the OR model a cycle is not enough: a deadlock requires that no reachable process can send a message.*

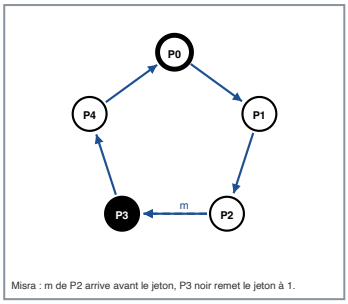
Schémas



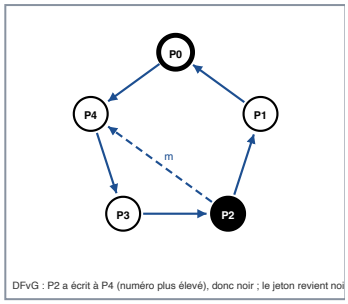
2PC : deux phases, carré = prêt.



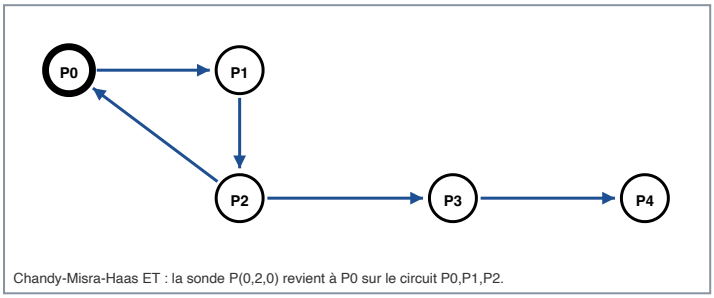
3PC : phase Prepare supplémentaire, état tampon p.



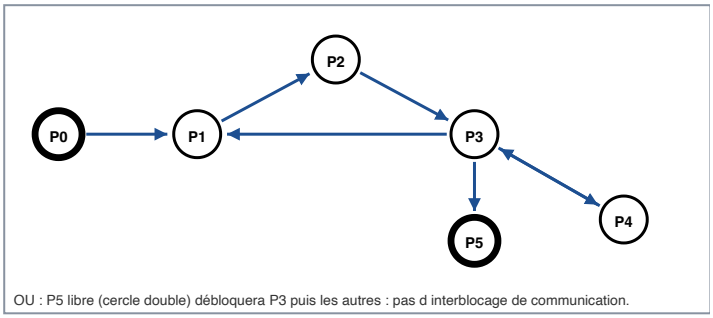
Misra.



DFvG.



Chandy-Misra-Haas ET.



Probe-echo OU.

Algorithmes clés

2PC

```
Coordinateur: log(Preparation) ; send ⟨Prepare⟩ à tous ;
collecter les votes
    tous Oui : log(Commit) ; send ⟨Commit⟩ ; sinon (un
Non ou timeout) : log(Abort) ; send ⟨Abort⟩
Participant: attendre ⟨Prepare⟩ (timeout : abort) ;
log(Ready Oui/Non) ; send vote ; attendre la décision ;
log ; appliquer ; send ⟨Done⟩
```

Dijkstra-Feijen-van Gasteren

```
R1: envoi à un numéro plus élevé (non initiateur) : état
:= noir
R2: noir qui envoie le jeton : jeton := noir ; blanc :
garde la couleur du jeton
R3: noir qui envoie le jeton : état := blanc
P0: jeton blanc et P0 blanc : terminé ; sinon nouveau
jeton blanc
```

Misra (anneau, FIFO)

```
Rp: { message applicatif reçu }  état := noir
Pi, jeton reçu (passif): si noir : token := 1 sinon
token := token + 1 ; état := blanc ; transmettre
P0, jeton reçu: si blanc et token = n : terminé
                sinon état := blanc ; transmettre
token := 1
```

Chandy-Misra-Haas ET (processus k)

```
Rp: { ⟨P(i, s, k)⟩ reçue }
    si w[k] et k ≠ i et ¬depend[k, i] : send ⟨P(i, k,
j)⟩ à chaque successeur j ; depend[k, i] := vrai
    si w[k] et k = i : k interbloqué
```

Exercices corrigés (chapitre 6)

TD ex. 1 : fausse terminaison avec les 4 compteurs

fait en cours, scénario reconstruit

Énoncé. Scénario où une seconde vague trop précoce valide une terminaison inexistante.

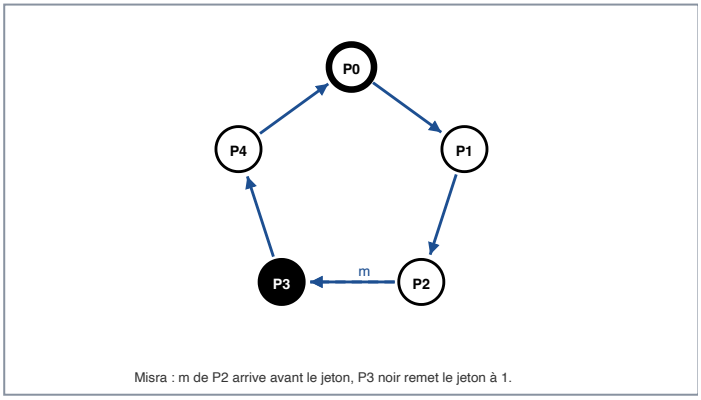
1. P1 envoie z à P0 (lent) : s1 = 1. La vague 1 lit P0 : (0,0). La vague 2, trop précoce, lit aussi P0 : (0,0).
2. P0 envoie y à P2 (après avoir été lu), P2 le reçoit : r2 = 1. Les deux vagues lisent P1 : (1,0) puis P2 : (0,1).
3. Chaque vague trouve S = 1, R = 1 : E1 = R1 = E2 = R2 = 1, terminaison proclamée alors que z est en transit. La cause est d2 < f1 ; avec f1 < d2 la vague 2 lit P0 après z et trouve S = R = 2 ≠ 1.

A ECRIRE La seconde vague doit démarrer après la fin de la première, sinon les deux voient la même coupe incohérente.
EN *The second wave must start after the first one has ended.*

TD ex. 2 : Misra (au plus deux tours, FIFO, passivité)

rédigé, schéma du prof

Énoncé. Prouver qu'au plus 2 tours suffisent après la terminaison. Faut-il traiter les messages en attente avant le jeton ? Et sans FIFO ?



m de P2 devant le jeton, P3 noir.

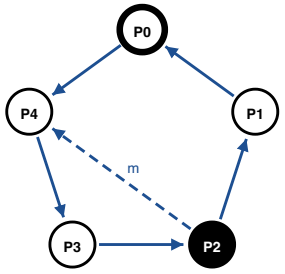
1. Après la terminaison plus personne ne devient noir. Le tour en cours peut échouer ; le premier tour démarré après blanchit tout le monde ; le suivant voit n blancs : **au plus 2 tours**.
2. Oui, traiter les messages en attente d'abord : un processus actif qui passerait le jeton serait faussement blanc.
3. Sans FIFO : faux, le jeton peut doubler un message en transit.

A ECRIRE Au plus deux tours ; jeton traité seulement passif ; FIFO indispensable.
EN *At most two rounds after termination; the token is processed only when passive.*

TD ex. 3 : Dijkstra-Feijen-van Gasteren

rédigé, schémas du prof

Énoncé. Un tour suffit quand ? Borne D ? Pourquoi échoue-t-il en asynchrone ? Routage par l'anneau ?



DFvG : P2 a écrit à P4 (numéro plus élevé), donc noir ; le jeton revient noir.

P2 noir après un envoi vers P4.

1. Un tour suffit si aucun processus n'a noirci (aucun envoi vers un numéro plus élevé).
2. Borne D : attendre D avant de traiter le jeton, puis regarder la file.
3. En asynchrone non borné : un message vers un numéro plus bas, lent, réactive un processus déjà vu passif sans noircir personne.
4. Routage par l'anneau seulement : FIFO fait doubler le jeton impossible ; différence avec Misra : **n n'a pas besoin d'être connu**, seule la couleur compte.

A ECRIRE Un tour sans envoi en arrière ; attendre D ; pas besoin de n.
EN *Waiting D guarantees that every message in transit has reached the mailbox.*

TD ex. 4 : sans n ni anneau, Dijkstra-Scholten

rédigé

Énoncé. Détection de terminaison sans connaître n ni un ordre sur un anneau, un seul initiateur.

1. Chaque message applicatif est acquitté. Le premier message reçu définit le parent (pas d'ack immédiat), les suivants sont acquittés tout de suite. Chaque processus compte ses messages non acquittés (déficit).
2. Un processus passif avec déficit 0 acquitte son parent et se désengage. L'initiateur passif avec déficit 0 déclare la terminaison.
3. Mode de transmission : asynchrone suffit (les acquittements remplacent le synchronisme), canaux fiables.

A ECRIRE Dijkstra-Scholten : arbre d'engagement, acks, déficit.
EN *Termination is detected when the root is passive and all its messages are acknowledged.*

TD ex. 5 : (R1 et R2) ou (R3 et R4), WFG et Chandy-Misra-Haas

rédigé, esquisse non prouvée

Énoncé. Représenter le WFG et dire si Chandy-Misra-Haas s'applique.

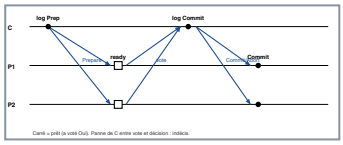
1. WFG : pour chaque processus, des nœuds de clause reliés par des arêtes OU, chaque clause reliée aux propriétaires de ses ressources par des arêtes ET.
2. Chandy-Misra-Haas ET donnerait de **faux interblocages** (le processus peut sortir par une autre clause). L'OU traite tous les successeurs comme alternatifs, ce qui est faux pour les groupes ET.
3. Alternative : un écho qui renvoie « interbloqué ou non » : interbloqué ssi pour chaque clause au moins un propriétaire est interbloqué ; un processus libre répond non.

A ECRIRE Non applicable tel quel ; un écho à valeur booléenne (ET sur les clauses, OU sur les ressources) convient.
EN *A cycle is not enough in the AND-OR model.*

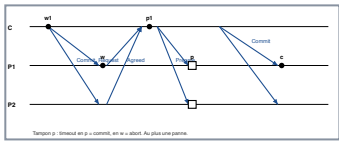
TD ex. 6 : 2PC contre 3PC

rédigé

Énoncé. En quel sens 3PC est-il non bloquant ?



2PC



3PC

1. 2PC : l'état d'attente a commit et abort dans son ensemble de concurrence, donc un survivant seul ne peut pas décider.
2. 3PC : l'état tampon p sépare l'attente du commit ; timeout en w : abort, en p : commit. Les survivants décident sans attendre le retour du coordinateur.
3. Hypothèses : au plus une panne, timeouts exacts, pas de partition ; trois aller-retours au lieu de deux.

A ECRIRE Non bloquant = une panne n'empêche pas les autres de décider.
EN *3PC adds a prepared state so that operational sites can decide on their own.*

Réponses types des annales

Q. Misra : combien de tours au plus après la terminaison ?

R. Deux. Après la terminaison plus personne ne noircit : le premier tour démarré après blanchit tous les processus (il peut échouer), le suivant voit n blancs consécutifs.

EN *At most two rounds: the first round started after termination whitens every process, the second sees n white processes.*

Q. Pourquoi 2PC est-il bloquant et pas 3PC ?

R. En 2PC, l'état w a commit et abort dans son ensemble de concurrence : un survivant seul ne peut décider. 3PC ajoute l'état tampon p : en p timeout vers commit, en w vers abort, sous l'hypothèse d'au plus une panne.

EN *In 2PC the waiting state may be concurrent with both commit and abort; 3PC adds a prepared state that removes this ambiguity.*

Q. Quatre compteurs : pourquoi deux vagues ?

R. Une vague isolée peut être une coupe incohérente (envoi non vu, réception non vue). Avec E1 = R1 = E2 = R2 et la seconde vague démarrée après la première, la preuve donne E(f1) = R(f1) : plus rien en transit.

EN *A single wave may observe an inconsistent cut; two waves with equal totals prove that nothing was in transit.*

Q. Détection d'interblocage ET contre OU

R. ET : sonde qui revient à l'initiateur sur un circuit. OU : probe-echo, l'initiateur est interbloqué s'il reçoit l'ack de tous ses successeurs ; un processus libre ignore la sonde.

EN *In the AND model a returning probe proves a cycle; in the OR model the initiator needs an echo from all its successors.*

Q. DFvG : pourquoi échoue-t-il en asynchrone ?

R. Un message envoyé vers un numéro plus bas laisse l'émetteur blanc ; s'il est lent il réactive un processus déjà vu passif sans noircir personne : fausse terminaison. Parade : attendre D (borne de délai) avant de traiter le jeton.

EN *A slow message sent to a lower-numbered process can reactivate it behind the token without any process turning black.*

PIEGES **à éviter**

- 2PC : un seul Non impose abort ; un participant « prêt » ne peut plus avorter seul.
- 3PC : non bloquant seulement avec au plus une panne et des timeouts exacts.
- Une vague de comptage S = R ne suffit pas : deux vagues, la seconde après la fin de la première.
- Misra : n doit être connu (token = n) et les canaux FIFO.
- DFvG : seul l'envoi vers un numéro plus élevé noircit.
- ET : l'initiateur doit être sur un circuit ; OU : un cycle ne suffit pas.

Annexe | exclusion mutuelle

TOMBE Dans quatre annales : Ricart-Agrawala avec preuves ME1 à ME3 (2011), token ring et Suzuki-Kasami modifiés pour un processus prioritaire (2012-13), double anneau à jeton (2013-14), Suzuki-Kasami avec requêtes concurrentes et jeton perdu (2017-18). Pas de page de cours en ligne : fiche tirée des slides ExclMut.pdf et 2021-DistAlgo_Exmut.pdf.

Explications

Le problème

Plusieurs processus veulent entrer dans une section critique : ME1 au plus un à la fois (safety), ME2 pas d'interblocage, ME3 pas de famine (liveness). Deux familles de solutions réparties : à permission (Lamport, Ricart-Agrawala : on demande à tous) et à jeton (anneau, Suzuki-Kasami : seul le détenteur entre).

Lamport contre Ricart-Agrawala

Lamport : requêtes estampillées, file locale, ack, release ; entrer quand ma requête est première et que tous ont répondu ; 3(N-1) messages, canaux FIFO. Ricart-Agrawala : pas de release, on retient la réponse tant qu'on est en CS ou qu'on a une estampille plus petite ; 2(N-1) messages, FIFO inutile.

À noter sur la feuille

Problème et propriétés

n processus Pi, graphe complet, messages. Chacun veut périodiquement : entrer en section critique (CS), l'exécuter, en sortir. Pourquoi : l'atomicité n'existe que jusqu'à un certain niveau (compte bancaire : load, add, store entrelacés : 2 entrelacements sur 12 donnent le bon résultat).

ME1 exclusion mutuelle : au plus un processus en CS (safety). **ME2** absence d'interblocage : au moins un processus éligible entre (liveness). **ME3** équité (fairness) : tout processus qui veut entrer finit par y arriver, pas de famine. Mesure : attente bornée, n-fairness ; **FIFO fairness** quand n = 0.

EN *ME1 is safety, ME2 and ME3 are liveness.*

Peterson (mémoire partagée, 2 processus)

interested[tid] := vrai ; **last_interested := tid** ; tant que (last_interested = tid ET interested[autre]) attendre. Sortie : interested[tid] := faux. Principe : je dis que je veux entrer, je dis que je suis le dernier, j'attends tant que l'autre veut aussi et que je suis encore le dernier.

ME2 (liveness) : par l'absurde, deux bloqués verraient chacun last_interested = soi, contradiction. ME1 (safety) : 4 événements par fil (WI, WL, RI, RL) ; on suppose que les deux entrent et on trouve un **cycle de happened-before**. Échanger les deux écritures casse la safety (pas de cycle) ; modèles mémoire faibles : réordonnancements, barrières (fences).

EN *Peterson: announce interest, give priority to the other, wait while the other is interested and you were the last to announce.*

Centralisé et naïf

Centralisé : un coordinateur, file des requêtes, req / reply / release. Défauts : point unique de panne, contention, pas d'équité FIFO (une requête arrivée tard par un chemin lent).

Naïf : diffuser REQUEST, attendre les ACK de tous, entrer, diffuser RELEASE. Ne marche pas : deux REQUEST concurrentes se confondent. Remède : estampiller les requêtes.

Lamport (canaux FIFO)

Chaque processus a une file Q des requêtes connues. **LA1** demander : envoyer une requête estampillée à tous, l'ajouter à sa file. **LA2** à la réception : l'ajouter à Q et envoyer un ACK estampillé (mais une fois sorti de la CS). **LA3** entrer quand ma requête est **première dans Q** (ordre chronologique) ET tous les ACK reçus. **LA4** sortir : retirer ma requête de Q, envoyer release estampillé. **LA5** à la réception d'un release : retirer la requête.

Observation : quand tous les ACK sont reçus, toute requête en route a une estampille plus grande, donc vue cohérente de la file. ME1 : si j et k entraient, Qj.ts < Qk.ts et Qk.ts < Qj.ts, impossible. ME2 : la chaîne d'attente est acyclique. ME3 : les nouvelles requêtes arrivent en fin de file. FIFO fairness : par FIFO, k a reçu la requête de j avant l'ack de j. **Messages : 3(N-1)** (requêtes + acks + releases).

EN *A process enters when its request is first in its local queue and it has received an acknowledgement from everyone.*

Ricart et Agrawala

1. Diffuser une requête estampillée. 2. À la réception d'une requête : **ack** si je ne veux pas entrer, ou si je veux entrer mais mon estampille est plus grande (la sienne est plus petite) ; sinon (déjà en CS ou estampille plus petite) ne rien répondre et **mémoriser la requête en attente**. 3. Entrer quand je reçois l'ack de tous. 4. À la sortie : envoyer l'ack aux requêtes en attente avant d'en refaire une. **Pas de release**.

Messages : 2(N-1) (N-1 requêtes + N-1 acks). ME3 (FIFO fairness) tient **même si les canaux ne sont pas FIFO**. Exercice du cours : L-exclusion (jusqu'à L processus en CS), modifier R-A.

EN *A process grants its permission unless it is in the critical section or it has a smaller timestamp; deferred replies are sent on exit.*

Jeton : anneau et naïf

Anneau à jeton : anneau logique pi → p(i+1) mod n ; seul le détenteur du jeton entre en CS ; sinon il le fait suivre. Safety et liveness garanties, **pas l'ordre** ; 1 message par sortie ; (N-1)-fairness ; délai entre sortie et entrée suivante entre 1 et N-1 messages.

Naïf (graphe complet) : le « roi » détient le jeton et la file des demandes ; à la sortie il envoie le jeton avec la file. Le roi change, donc chaque demande est diffusée. Exercice : deux scénarios qui échouent (une demande jamais satisfaite : famine ; un processus reçoit le jeton sans l'avoir demandé).

Suzuki-Kasami

req[j] : numéro de séquence de la dernière requête de j (tout le monde). **last[j]** : numéro de la dernière visite de j en CS (dans le jeton). Jeton = (Q, last). Demande : req[j]++ puis diffuser (j, req[j]) ; attendre le jeton. Réception de (k, num) : req[k] := max(req[k], num) (écarte les requêtes périmées).

Sortie : last[j] := req[j] ; pour tout $k \neq j$, $k \notin Q$ et req[k] = last[k] + 1 : ajouter k à Q ; si Q non vide : envoyer (queue de Q, last) à la tête de Q. Au plus N messages par entrée (N-1 requêtes + le jeton), 0 si on détient déjà le jeton.

EN *The token carries the queue and the last-served numbers; a request is fresh only if req[k] = last[k] + 1.*

Algorithmes clés

Ricart-Agrawala (Pi)

```
demander : ts := horloge ; send ⟨REQUEST, ts, i⟩ à tous
; attendre ⟨ACK⟩ de tous ; entrer en CS
Rp: { ⟨REQUEST, ts, j⟩ reçu }
    si (pas en CS et ne veut pas) ou (veut mais (ts, j)
< (mon ts, i)) alors send ⟨ACK⟩ à j
    sinon pending := pending ∪ {j}
sortie : send ⟨ACK⟩ à chaque j ∈ pending ; pending := ∅
```

Suzuki-Kasami (Pj)

```
init : pour tout i : req[i] := last[i] := 0
entrée : req[j] := req[j] + 1 ; send ⟨j, req[j]⟩ à tous
; attendre le jeton (Q, last) ; CS
sortie : last[j] := req[j] ; pour tout k ≠ j : si k ∉ Q
et req[k] = last[k] + 1 : Q := Q + k
        si Q non vide : send (queue(Q), last) à tête(Q)
Rp: { ⟨k, num⟩ reçu }  req[k] := max(req[k], num)
```

Lamport (Pi)

```
demander : send ⟨REQUEST, ts⟩ à tous ; Q := Q + (ts, i)
Rp: { ⟨REQUEST, ts, j⟩ }  Q := Q + (ts, j) ; send ⟨ACK⟩
à j
entrer : (ts, i) = tête(Q) et ACK de tous reçus
sortie : Q := Q − (ts, i) ; send ⟨RELEASE⟩ à tous    Rp
⟨RELEASE, j⟩ : Q := Q − (·, j)
```

Exercices corrigés (chapitre 7)

Compteur partagé et valeurs possibles slides

Énoncé. Deux appels f() parallèles, chacun incrémente c cent fois sans protection. Valeurs finales possibles ?

1. Chaque c = c + 1 est une lecture suivie d'une écriture ; entrelacées, elles perdent des incréments.
2. Valeur maximale : 2N (chaque lecture suivie de son écriture). Valeur minimale : **2** (le cours construit un entrelacement qui écrase presque tout).
3. Donc c est entre 2 et 2N inclus.

A ECRIRE c ∈ [2, 2N].
EN *The final value lies between 2 and 2N.*

Peterson : pourquoi l'ordre des écritures compte slides

Énoncé. Que se passe-t-il si on échange les deux écritures (last_interested puis interested) ?

1. La preuve de ME1 suppose l'ordre WI, WL, RI, RL pour chaque fil et construit un cycle de happened-before si les deux entrent.
2. En échangeant les écritures, le cycle disparaît : la preuve ne marche plus et la safety est **violée** (le slide : plus de cycle).
3. Les modèles mémoire faibles réordonnent aussi les accès : il faut des barrières (fences).

A ECRIRE L'échange casse ME1 ; le cycle de happened-before n'existe plus.
EN *Swapping the two writes breaks the safety proof: there is no longer a happens-before cycle.*

Réponses types des annales

Q. Prouver ME1 pour Ricart-Agrawala (exam 2011)

R. Par l'absurde : si Pj et Pk sont en CS, chacun a reçu l'ack de l'autre pour sa propre requête. Avec TS(j) < TS(k), Pj n'a pas ack Pk tant qu'il veut entrer ou est en CS (estampille plus petite) : Pk n'aurait pas pu entrer. Contradiction.

EN *If both were in the critical section, the one with the larger timestamp would have received an ack from the other, which holds back its reply.*

Q. Ricart-Agrawala : pourquoi pas de blocage (ME2) ?

R. La relation d'attente suit l'ordre des estampilles : une requête n'attend que des requêtes d'estampille plus petite (les plus grandes reçoivent un ack), donc pas de cycle ; la plus petite entre.

EN *Requests only wait for smaller timestamps, so the waiting relation is acyclic.*

Q. Complexités en messages par entrée en CS

R. Lamport 3(N-1) ; Ricart-Agrawala 2(N-1) ; anneau à jeton 1 message par sortie (attente jusqu'à N-1) ; Suzuki-Kasami au plus N.

EN *Lamport 3(N-1), Ricart-Agrawala 2(N-1), Suzuki-Kasami at most N messages.*

PIEGES **à éviter**

- Peterson : l'ordre des deux écritures compte (last_interested après interested), sinon safety violée.
- Lamport exige FIFO ; Ricart-Agrawala non.
- Ricart-Agrawala : pas de message release, les acks en attente partent à la sortie.
- Suzuki-Kasami : une requête est fraîche seulement si req[k] = last[k] + 1.
- Anneau à jeton : pas d'ordre d'entrée garanti, et un jeton perdu bloque tout (détection et régénération à prévoir).

Références et annales

Cours en ligne : **cours.lsro.fr/algodist** (chapitres 1 à 6, QCM, exercices corrigés) et cours.lsro.fr/algodist/revision (cette synthèse, version en ligne).

Annales à consulter (PDF du dossier Moodle, non republiés ici) : exam2011, exam2012, exam2012_2013, exam2013_2014, exam2017_2018, exam2019, CorrecExoFT-Exam2022. Moodle contient les exercices et leurs corrigés.

L'annexe sur l'exclusion mutuelle est tirée des slides 2021-DistAlgo_Exmut.pdf (le cours en ligne n'a pas encore de page sur ce chapitre) ; ExclMut.pdf est à relire dans les slides imprimés.